ICII CHART

	0/8	1/9	2/A	3/B	4/C	5/D	6/E	7/F
0	NUL	DLE		0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	FI	US	/	?	O	_	o	DEL

PLE II I/O ADDRESSES

ex	Decimal	Function
000	−16384	Keyboard Input
010	−16368	Clear Keyboard Strobe
020	−16352	Cassette Output Toggle
030	−16336	Speaker Toggle
040	−16320	Utility Strobe
050	−16304	Set Graphics Mode
051	−16303	Set Text Mode
052	−16302	Set All Text or All Graphics
053	−16301	Set Mixed Text and Graphics
054	−16300	Display Primary Page
055	−16299	Display Secondary Page
056	−16298	Display Lo-Res Graphics
057	−16297	Display Hi-Res Graphics
058, 9	−16296, −16295	AN0 off, on
05A, B	−16294, −16293	AN1 off, on
05C, D	−16292, −16291	AN2 off, on
05E, F	−16290, −16289	AN3 off, on
060	−16288	Cassette Input
061-3	−16287 to −16285	PB1, PB2, PB3 Inputs
064-7	−16284 to −16281	Paddle 0 through 3 Inputs
070	−16272	Paddle Strobe
080-F	−16256 to −16241	Slot 0 I/O Space
090-F	−16240 to −16225	Slot 1 I/O Space
0A0-F	−16224 to −16209	Slot 2 I/O Space
0B0-F	−16208 to −16193	Slot 3 I/O Space
0C0-F	−16192 to −16177	Slot 4 I/O Space
0D0-F	−16176 to −16161	Slot 5 I/O Space
0E0-F	−16160 to −16145	Slot 6 I/O Space
0F0-F	−16144 to −16129	Slot 7 I/O Space
100-FF	−16128 to −15873	Slot 1 ROM Space
200-FF	−15872 to −15617	Slot 2 ROM Space
300-FF	−15616 to −15361	Slot 3 ROM Space
400-FF	−15360 to −15105	Slot 4 ROM Space
500-FF	−15104 to −14849	Slot 5 ROM Space
600-FF	−14848 to −14593	Slot 6 ROM Space
700-FF	−14592 to −14337	Slot 7 ROM Space
C800-CFFF	−14336 to −12289	Expansion ROM Space
CFFF	−12289	Switch for Expansion ROM

$3.50

S-C Macro Assembler

"Makes assembly language programming on the Apple as easy as programming in BASIC."

Programmer Reference Card

 February, 1982

S-C Software Corporation
2331 Gus Thommasson,
Suite 125
P.O. Box 280300
Dallas, Texas 75228
(214) 324-2050

S-C MACRO ASSEMBLER COMMANDS

ASM	Assemble source program.
AUTo AUTo num	Turn on automatic line numbering. Turn on automatic line numbering starting with num.
COPy #1, #2, #3	Insert a copy of line #1 through line #2 just before line #3.
DELete linenum	Delete specified line(s).
EDIt string linenum	Edit specified line(s).
FASt	Select normal listing speed.
FINd string linenum	List all lines containing the specified string.
HIDe	Use with LOAD and MERGE commands to join two source programs.
IL PROG NASCOSTO SEGUE QUELLO CARICATO INCrement value	Set auto-line-number increment to specified value.
LISt string linenum	List specified line(s).
LOAd	Load source program from tape.
LOAD filename	[DOS Command] Load source program from disk.
MANual	Turn off automatic line numbering
MEMory	Display memory pointers to source program and symbol table.
MERge	Use with HIDE and LOAD commands to join two source programs.
MGO expression	Execute object program, starting at address specified by value of expression.
MNTR	Enter system monitor.
NEW	Delete entire source program, and start all over.
PRT	Call user printer software through vector at $1009.
RENumber	Renumber all lines of the source program: number the first line 1000, and use an increment of 10.
RENumber #	Renumber all lines of the source program: number the first line #, and use an increment of 10.
RENumber #1, #2	Renumber all lines of the source program: number the first line #1, and use an increment of #2.
RENumber #1, #2, #3	Renumber from line #3 through the end of the source program: renumber line #3 as #1, and use an increment of #2.
REPlace /stra/strb/	Replace all occurences of stringa with stringb.
REStore	Restore root program after an aborted assembly.
RST expression	Set RESET vector to expression.
SAVe	Save source program on tape.
SAVE filename	[DOS Command] Save source program on disk.
SLOW	Select slow listing speed.
TEXt filename	Write source program to DOS text file with no line numbers.
TEXt # filename	Write source program to DOS text file, with line numbers.
TEXt/ filename	Write source program to DOS text file, with <ctrl-I>'s.
SYMbol	Display symbol table.
USR	User defined command, calls $1006.
VAL expression	Display value of expression.

Linenumber Parameters: "linenum" has the following options:

(blank)	All lines
num1	That line only
num1,num2	All lines from num1 through num2
num1,	All lines from num1 through end
,num1	All lines from beginning through num1

The numbers num1 and num2 may be line numbers, or a period; period signifies the line number of the last line entered or deleted.

String Parameters: "string" as a parameter in a command means "dstringd", where "d" is any printing character except comma (,), period(.), or a digit (0-9). The string may also contain a "wildcard" character (control-W).

EDIT MODE COMMANDS

When the EDIT command is being used, the following sub-commands are available to modify the line being edited.

control-B	move cursor back to beginning of the label field.
control-D	delete character under cursor.
control-Fx	move cursor to next occurence of "x" in line (if any).
control-H	(left arrow) move cursor left.
control-I	begin insertion mode; characters will be inserted until another control character is typed.
control-L	store current edited line and start editing the next line.
control-M	(RETURN) store the edited line.
control-N	move cursor to end of line.
control-O	begin insertion mode, but allow next character typed to be inserted even if it is a control-character.
control-Q	finish edit mode, chopping off all characters from cursor to end of line.
control-R	restore the original line without leaving edit-mode.
control-T	move cursor to next tab stop.
control-U	(right arrow) move cursor right.
control-X	abort the EDIT command.
control-@	erase from cursor to end of line without leaving edit-mode.

S-C MACRO ASSEMBLER DIRECTIVES

.AS dstringd	Ascii String (d is delimiter)
.AT dstringd	Ascii Terminated
.BS expression	Block Storage
.DA exprlist	DAta
.DO expression	Assemble block if expression true
.ELSE	Toggle .DO expression value
.EM	End Macro definition
.EN	ENd of program or included section
.EQ expression	EQuate expression to label
.FIN	FINish conditional assembly
.HS hexstring	Hex String
.IN filename	INclude a source file
.LIST MOFF	Macro LISTing OFF
.LIST MON	Macro LISTing ON
.LIST OFF	LISTing OFF
.LIST ON	LISTing ON
.MA macro name	MAcro definition
.OR expression	ORigin
.PG	PaGe eject
.TA expression	Target Address
.TF filename	Target File
.TI expression,title	TItle
.US whatever	USer defined

Expressions in the operand field of directives or instructions are evaluated from left to right. Operands may be labels, decimal numbers, hexadecimal numbers, literal ASCII characters, or *. Operators may be arithmetic (+, -, *, or /) or relational (<, = or >).

The .DA directive may be followed by one or more expressions separated by commas. Each expression may be one of these forms:

expression	Two bytes, low-byte first.
#expression	Low-order byte only.
/expression	High-order byte only.

SWEET-16 OPCODES

The standard version of SWEET-16 is invoked by the 6502 instruction "JSR $F689"; the bytes immediately following contain opcodes for SWEET-16 to process. SWEET-16 opcodes will be executed until the "RTN" opcode, which returns to 6502 mode.

00	RTN		Return to 6502 code.
01 ea	BR	addr	Unconditional Branch.
02 ea	BNC	addr	Branch if Carry = 0.
03 ea	BC	addr	Branch if Carry = 1.
04 ea	BP	addr	Branch if last result positive.
05 ea	BM	addr	Branch if last result negative.
06 ea	BZ	addr	Branch if last result zero.
07 ea	BNZ	addr	Branch if last result non-zero.
08 ea	BM1	addr	Branch if last result = −1.
09 ea	BNM1	addr	Branch if last result not −1.
0A	BK		Execute 6502 BRK instruction.
0B	RS		Return from SWEET-16 subroutine.
0C ea	BS	addr	Call SWEET-16 subroutine.
1n lo hi	SET	n, value	Rn <-- value.
2n	LD	n	RO <-- (Rn).
3n	ST	n	Rn <-- (RO).
4n	LD	@n	MA = (Rn), ROL <-- (MA), Rn <-- MA+1, ROH <-- 0.
5n	ST	@n	MA = (Rn), MA <-- (ROL), Rn <-- MA+1.
6n	LDD	@n	MA = (Rn), Rn <-- (MA, MA+1), Rn <-- MA+2.
7n	STD	@n	MA = (Rn), MA, MA+1 <-- (RO), Rn <-- MA+2.
8n	POP	@n	MA = (Rn)−1, ROL <-- (MA), ROH <-- 0, Rn <-- MA.
9n	STP	@n	MA <-- (Rn)−1, MA <-- ROL, Rn <-- MA.
An	ADD	n	RO <-- (RO) + (Rn).
Bn	SUB	n	RO <-- (RO) − (Rn).
Cn	POPD	@n	MA = (Rn)−2, MA, MA+1 <-- RO, Rn <-- MA.
Dn	CPR	n	R13 <-- (RO) − (Rn), R14 <-- status flags.
En	INR	n	Rn <-- (Rn) + 1.
Fn	DCR	n	Rn <-- (Rn) − 1.

DOS COMMANDS RELEVANT TO S-C MACRO ASSEMBLER

BLOAD name (,Aaddr)(,Ss)(,Dd)(,Vv)
BRUN name (,Aaddr)(,Ss)(,Dd)(,Vv)
BSAVE name ,Aaddr, Llength (,Ss)(,Dd)(,Vv)
CATALOG
DELETE name (,Ss)(,Dd)(,Vv)
FP
IN#slot
INT
LOAD name (,Ss)(,Dd)(,Vv)
LOCK name (,Ss)(,Dd)(,Vv)
MON (C)(I)(O)
NOMON (C)(I)(O)
PR#slot
RENAME oldname,newname (,Ss)(,Dd)(,Vv)
SAVE name (,Ss)(,Dd)(,Vv)
UNLOCK name (,Ss)(,Dd)(,Vv)
VERIFY name (,Ss)(,Dd)(,Vv)

APPLE II MONITOR COMMANDS

All Apple II Monitor commands are usable from inside S-C Macro Assembler. Type a dollar sign ($) and the monitor command you wish to use.

$addr	Examine one location
$addr.addr	Display range of locations
$ (null command)	Display next (up to) eight locations
$addr:val val ...	Store values in memory
$:val val ...	Continue storing values in memory
$addr<addr.addrM	Move block of memory
$addr<addr.addrV	Verify block of memory
$addr.addrR	Read cassette tape
$addr.addrW	Write cassette tape
$addrG	Execute program
$addrL	Dis-assemble 20 lines
$E (control-E)	Display 6502 registers
$I	Set inverse display mode
$N	Set normal display mode
$val+val	Add two values, print result
$val−val	Subtract value, print result
$Y (control-Y)	Jump to $03F8
$B (control-B)	Enter language in ROM at $E000
$C (control-C)	Enter language in ROM at $E003
$slotP (control-P)	Direct output to slot
$slotK (control-K)	Accept input from slot

The following are not available in the Autostart ROM:

$addrS	Execute one 6502 instruction, at addr. If no addr specified, execute next instruction.
$addrT	Trace execution starting at addr. If no addr specified, start at next instruction.

PAGE THREE LOCATIONS

Address	Normal Contents	Function
$3D0-3D2	4C BF 9D	Warmstart DOS
$3D3-3D5	4C 84 9D	Coldstart DOS
$3D6-3D8	4C FD AA	Enter File Manager
$3D9-3DB	4C B5 B7	Enter RWTS
$3DC-3E2	AD OF 9D AC OE 9D 60	Find File Manager Parmlist
$3E3-3E9	AD C2 AA AC C1 AA 60	Find RWTS IOB
$3EA-3EC	4C 51 A8	Rehook DOS Intercepts
$3ED-3EE	EA EA	Not Used
$3EF-3F1	4C 59 FA	Handle BRK Instruction
$3F2-3F3	BF 9D	RESET Vector (Autostart ROM)
$3F4	38	Power-up Byte $3F3 EOR #$A5
$3F5-3F7	4C 58 FF	Handle & (Applesoft)
$3F8-3FA	4C 65 FF	Handle CTRL-Y (Monitor)
$3FB-3FD	4C 65 FF	Handle NMI
$3FE-3FF	65 FF	IRQ Vector

INSTRUCTIONS WITH OPCODE AND CUTION CYCLES

	Branch if Plus	(Branch if N=0)	10
	Branch if Minus	(Branch if N=1)	30
	Branch if No Overflow	(Branch if V=0)	50
	Branch if Overflow	(Branch if V=1)	70
CC	Branch if Carry Clear	(Branch if C=0)	90
CS	Branch if Carry Set	(Branch if C=1)	B0
NE	Branch if Not Equal	(Branch if Z=0)	D0
EQ	Branch if Equal	(Branch if Z=1)	F0

2 cycles if no branch; 3 cycles if branch into same page; 4 cycles if branch into different page.

MP INSTRUCTIONS

P	Unconditional Jump	Absolute	4C 3
P	Unconditional Jump	Indirect	6C 5
	Jump to Subroutine		20 6

PLIED ADDRESS MODE INSTRUCTIONS

RK	Break (Set B=1, Generate IRQ Interrupt)		00	7	.	.	B	.	I	.	.
LC	Clear Carry	C <-- 0	18	2	.	.	.	.	.	.	C
LD	Clear Decimal Mode	D <-- 0	D8	2	.	.	.	.	D	.	.
	Clear Interrupt Mask	I <-- 0	58	2	.	.	.	.	I	.	.
	Clear Overflow Status	V <-- 0	B8	2	.	V	.	.	.	.	.
DEX	Decrement X-register	X <-- (X) − 1	CA	2	N	.	.	.	.	Z	.
DEY	Decrement Y-register	Y <-- (Y) − 1	88	2	N	.	.	.	.	Z	.
DEY	Decrement Y-register	Y <-- (Y) − 1	88	2	N	.	.	.	.	Z	.
INX	Increment X-register	X <-- (X) + 1	E8	2	N	.	.	.	.	Z	.
INY	Increment Y-register	Y <-- (Y) + 1	C8	2	N	.	.	.	.	Z	.
NOP	No Operation	Does Nothing	EA	2	.	.	.	.	.	.	.
PHA	Push A-register on stack	M(S) <-- (A), S <-- (S) − 1	48	3	.	.	.	.	.	.	.
PHP	Push P-register on stack	M(S) <-- (P), S <-- (S) − 1	08	3	.	.	.	.	.	.	.
PLA	Pull Stack to A-register	S <-- (S) + 1, A <-- (M(S))	68	4	N	.	.	.	.	Z	.
PLP	Pull Stack to P-register	S <-- (S) + 1, A <-- (M(S))	28	4	restored						
RTI	Return from Interrupt	Pull to P and PC	40	6	restored						
RTS	Return from Subroutine	Pull to PC	60	6	.	.	.	.	.	.	.
SEC	Set Carry	C <-- 1	38	2	.	.	.	.	.	.	C
SED	Set Decimal Mode	D <-- 1	F8	2	.	.	.	.	D	.	.
SEI	Set Interrupt Mask	I <-- 1	78	2	.	.	.	.	I	.	.
TAX	Transfer A to X	X <-- (A)	AA	2	N	.	.	.	.	Z	.
TAY	Transfer A to Y	Y <-- (A)	A8	2	N	.	.	.	.	Z	.
TSX	Transfer S to X	X <-- (S)	BA	2	N	.	.	.	.	Z	.
TXA	Transfer X to A	A <-- (X)	8A	2	N	.	.	.	.	Z	.
TXS	Transfer X to S	S <-- (X)	9A	2	.	.	.	.	.	.	.
TYA	Transfer Y to A	A <-- (Y)	98	2	N	.	.	.	.	Z	.

STATUS REGISTER

7	6	5	4	3	2	1	0
N	V	*	B	D	I	Z	C

Bit	Value = 0	Value = 1
N	Negative Last result + ($00-$7F).	Last result − ($80−$FF).
V	Overflow Last result no overflow, and after CLV.	Last result overflow.
*	Unused Never.	Always.
B	Break	After BRK.
D	Decimal After CLD (Binary mode).	After SED (Decimal mode).
I	Interrupt After CLI (IRQ enabled).	After SEI (IRQ disabled).
Z	Zero Last result non-zero.	Last result zero.
C	Carry Last result no Carry, and after CLC.	Last result did carry, and after SEC.

6502 INSTRUCTIONS WITH OPCODE AND EXECUTION CYCLES

		Accumulator	Immediate	Zero Page	Absolute	Zero Page, X	Absolute, X	Zero Page, Y	Absolute, Y	(Zero Page, X)	(Zero Page), Y		
ADC	Add with Carry	--	69	65	6D	75	7D	--	79	61	71	N V	ZC
	A <-- (A) + (M) + (C)	--	2	3	4	4	*4	--	*4	6	*5		
AND	Logical "And"	--	29	25	2D	35	3D	--	39	21	31	N	Z .
	A <-- (A) and (M)	--	2	3	4	4	*4	--	*4	6	*5		
ASL	Shift Byte Left	0A	--	06	0E	16	1E	--	--	--	--	N	ZC
	C <-- [7.....0] <-- 0	2	--	5	6	6	7	--	--	--	--		
BIT	Test Bits in Memory	--	--	24	2C	--	--	--	--	--	--	N V	Z .
	Z <-- (A) and (M), N <-- (M) bit 7, V <-- (M) bit 6	--	--	3	4	--	--	--	--	--	--		
CMP	Compare with A-register	--	C9	C5	CD	D5	DD	--	D9	C1	D1	N	ZC
	N, Z, C <-- (A) − (M)	--	2	3	4	4	*4	--	*4	6	*5		
CPX	Compare with X-register	--	E0	E4	EC	--	--	--	--	--	--	N	ZC
	N, Z, C <-- (X) − (M)	--	2	3	4	--	--	--	--	--	--		
CPY	Compare with Y-register	--	C0	C4	CC	--	--	--	--	--	--	N	ZC
	N, Z, C <-- (Y) − (M)	--	2	3	4	--	--	--	--	--	--		
DEC	Decrement Memory Byte	--	--	C6	CE	D6	DE	--	--	--	--	N	Z .
	M <-- (M) − 1	--	--	5	6	6	7	--	--	--	--		
EOR	Exclusive-Or	--	49	45	4D	55	5D	--	59	41	51	N	Z .
	A <-- (A) eor (M)	--	2	3	4	4	*4	--	*4	6	*5		
INC	Increment Memory Byte	--	--	E6	EE	F6	FE	--	--	--	--	N	Z .
	M <-- (M) + 1	--	--	5	6	6	7	--	--	--	--		
LDA	Load A-register	--	A9	A5	AD	B5	BD	--	B9	A1	B1	N	Z .
	A <-- (M)	--	2	3	4	4	*4	--	*4	6	*5		
LDX	Load X-register	--	A2	A6	AE	--	--	B6	BE	--	--	N	Z .
	X <-- (M)	--	2	3	4	--	--	4	*4	--	--		
LDY	Load Y-register	--	A0	A4	AC	B4	BC	--	--	--	--	N	Z .
	Y <-- (M)	--	2	3	4	4	*4	--	--	--	--		
LSR	Shift Byte Right	4A	--	46	4E	56	5E	--	--	--	--	N	ZC
	0 --> [7.....0] --> C	2	--	5	6	6	7	--	--	--	--		
ORA	Inclusive-Or	--	09	05	0D	15	1D	--	19	01	11	N	Z .
	A <-- (A) or (M)	--	2	3	4	4	*4	--	*4	6	*5		
ROL	Roll Byte Left	2A	--	26	2E	36	3E	--	--	--	--	N	ZC
	C <-- [7.....0] <-- C	2	--	5	6	6	7	--	--	--	--		
ROR	Roll Byte Right	6A	--	66	6E	76	7E	--	--	--	--	N	ZC
	C --> [7.....0] --> C	2	--	5	6	6	7	--	--	--	--		
SBC	Subtract with Borrow	--	E9	E5	ED	F5	FD	--	F9	E1	F1	N V	ZC
	A <-- (A)-(M) + (C)-1	--	2	3	4	4	*4	--	*4	6	*5		
STA	Store A-register	--	--	85	8D	95	9D	--	99	81	91	. .	. .
	M <-- (A)	--	--	3	4	4	5	--	5	6	6		
STX	Store X-register	--	--	86	8E	--	--	96	--	--	--	. .	. .
	M <-- (X)	--	--	3	4	--	--	4	--	--	--		
STY	Store Y-register	--	--	84	8C	94	--	--	--	--	--	. .	. .
	M <-- (Y)	--	--	3	4	4	--	--	--	--	--		

*Add one cycle if indexing causes a page boundary to be crossed.

$3.50

ASCII CHART

	0/8	1/9	2/A	3/B	4/C	5/D	6/E	7/F
0	NUL	DLE		0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	FI	US	/	?	O	_	o	DEL

APPLE II I/O ADDRESSES

Hex	Decimal	Function
$C000	−16384	Keyboard Input
$C010	−16368	Clear Keyboard Strobe
$C020	−16352	Cassette Output Toggle
$C030	−16336	Speaker Toggle
$C040	−16320	Utility Strobe
$C050	−16304	Set Graphics Mode
$C051	−16303	Set Text Mode
$C052	−16302	Set All Text or All Graphics
$C053	−16301	Set Mixed Text and Graphics
$C054	−16300	Display Primary Page
$C055	−16299	Display Secondary Page
$C056	−16298	Display Lo-Res Graphics
$C057	−16297	Display Hi-Res Graphics
$C058, 9	−16296, −16295	AN0 off, on
$C05A, B	−16294, −16293	AN1 off, on
$C05C, D	−16292, −16291	AN2 off, on
$C05E, F	−16290, −16289	AN3 off, on
$C060	−16288	Cassette Input
$C061-3	−16287 to −16285	PB1, PB2, PB3 Inputs
$C064-7	−16284 to −16281	Paddle 0 through 3 Inputs
$C070	−16272	Paddle Strobe
$C080-F	−16256 to −16241	Slot 0 I/O Space
$C090-F	−16240 to −16225	Slot 1 I/O Space
$C0A0-F	−16224 to −16209	Slot 2 I/O Space
$C0B0-F	−16208 to −16193	Slot 3 I/O Space
$C0C0-F	−16192 to −16177	Slot 4 I/O Space
$C0D0-F	−16176 to −16161	Slot 5 I/O Space
$C0E0-F	−16160 to −16145	Slot 6 I/O Space
$C0F0-F	−16144 to −16129	Slot 7 I/O Space
$C100-FF	−16128 to −15873	Slot 1 ROM Space
$C200-FF	−15872 to −15617	Slot 2 ROM Space
$C300-FF	−15616 to −15361	Slot 3 ROM Space
$C400-FF	−15360 to −15105	Slot 4 ROM Space
$C500-FF	−15104 to −14849	Slot 5 ROM Space
$C600-FF	−14848 to −14593	Slot 6 ROM Space
$C700-FF	−14592 to −14337	Slot 7 ROM Space
$C800-CFFF	−14336 to −12289	Expansion ROM Space
$CFFF	−12289	Switch for Expansion ROM

S-C Macro Assembler

"Makes assembly language programming on the Apple as easy as programming in BASIC."

Programmer Reference Card

 February, 1982

S-C Software Corporation
2331 Gus Thommasson,
Suite 125
P.O. Box 280300
Dallas, Texas 75228
(214) 324-2050

S-C MACRO ASSEMBLER COMMANDS

ASM	Assemble source program.
AUTo AUTo num	Turn on automatic line numbering. Turn on automatic line numbering starting with num.
COPy #1, #2, #3	Insert a copy of line #1 through line #2 just before line #3.
DELete linenum	Delete specified line(s).
EDIt string linenum	Edit specified line(s).
FASt	Select normal listing speed.
FINd string linenum	List all lines containing the specified string.
HIDe	Use with LOAD and MERGE commands to join two source programs.
IL PROG NASCOSTO SEGUE QUELLO CARICATO INCrement value	Set auto-line-number increment to specified value.
LISt string linenum	List specified line(s).
LOAd	Load source program from tape.
LOAD filename	[DOS Command] Load source program from disk.
MANual	Turn off automatic line numbering
MEMory	Display memory pointers to source program and symbol table.
MERge	Use with HIDE and LOAD commands to join two source programs.
MGO expression	Execute object program, starting at address specified by value of expression.
MNTR	Enter system monitor.
NEW	Delete entire source program, and start all over.
PRT	Call user printer software through vector at $1009.
RENumber	Renumber all lines of the source program: number the first line 1000, and use an increment of 10.
RENumber #	Renumber all lines of the source program: number the first line #, and use an increment of 10.
RENumber #1, #2	Renumber all lines of the source program: number the first line #1, and use an increment of #2.
RENumber #1, #2, #3	Renumber from line #3 through the end of the source program: renumber line #3 as #1, and use an increment of #2.
REPlace /stra/strb/	Replace all occurences of stringa with stringb.
RESTore	Restore root program after an aborted assembly.
RST expression	Set RESET vector to expression.
SAVe	Save source program on tape.
SAVE filename	[DOS Command] Save source program on disk.
SLOW	Select slow listing speed.
TEXt filename	Write source program to DOS text file with no line numbers.
TEXt # filename	Write source program to DOS text file, with line numbers.
TEXt/ filename	Write source program to DOS text file, with <ctrl-I>'s.
SYMbol	Display symbol table.
USR	User defined command, calls $1006.
VAL expression	Display value of expression.

Linenumber Parameters: "linenum" has the following options:

(blank)	All lines
num1	That line only
num1,num2	All lines from num1 through num2
num1,	All lines from num1 through end
,num1	All lines from beginning through num1

The numbers num1 and num2 may be line numbers, or a period; period signifies the line number of the last line entered or deleted.

String Parameters: "string" as a parameter in a command means "dstringd", where "d" is any printing character except comma (,), period(.), or a digit (0-9). The string may also contain a "wildcard" character (control-W).

EDIT MODE COMMANDS

When the EDIT command is being used, the following sub-commands are available to modify the line being edited.

control-B	move cursor back to beginning of the label field.
control-D	delete character under cursor.
control-Fx	move cursor to next occurence of "x" in line (if any).
control-H	(left arrow) move cursor left.
control-I	begin insertion mode; characters will be inserted until another control character is typed.
control-L	store current edited line and start editing the next line.
control-M	(RETURN) store the edited line.
control-N	move cursor to end of line.
control-O	begin insertion mode, but allow next character typed to be inserted even if it is a control-character.
control-Q	finish edit mode, chopping off all characters from cursor to end of line.
control-R	restore the original line without leaving edit-mode.
control-T	move cursor to next tab stop.
control-U	(right arrow) move cursor right.
control-X	abort the EDIT command.
control-@	erase from cursor to end of line without leaving edit-mode.

INSTRUCTIONS WITH OPCODE AND CUTION CYCLES

	Branch if Plus	(Branch if N=0)	10
	Branch if Minus	(Branch if N=1)	30
	Branch if No Overflow	(Branch if V=0)	50
	Branch if Overflow	(Branch if V=1)	70
C	Branch if Carry Clear	(Branch if C=0)	90
S	Branch if Carry Set	(Branch if C=1)	B0
NE	Branch if Not Equal	(Branch if Z=0)	D0
EQ	Equal	(Branch if Z=1)	F0

2 cycles if no branch; 3 cycles if branch into same page; 4 cycles if branch into different page.

MP INSTRUCTIONS

P	Unconditional Jump	Absolute	4C 3
P	Unconditional Jump	Indirect	6C 5
	Jump to Subroutine		20 6

PLIED ADDRESS MODE INSTRUCTIONS

RK	Break (Set B=1, Generate IRQ Interrupt)		00	7	.	.	.	B	.	I	.	.
LC	Clear Carry	C <-- 0	18	2	.	.	.	.	.	.	.	C
LD	Clear Decimal Mode	D <-- 0	D8	2	.	.	.	.	D	.	.	.
LI	Clear Interrupt Mask	I <-- 0	58	2	.	.	.	.	.	I	.	.
LV	Clear Overflow Status	V <-- 0	B8	2	.	V	.	.	.	.	.	.
DEX	Decrement X-register	X <-- (X) − 1	CA	2	N	.	.	.	.	.	Z	.
DEY	Decrement Y-register	Y <-- (Y) − 1	88	2	N	.	.	.	.	.	Z	.
DEY	Decrement Y-register	Y <-- (Y) − 1	88	2	N	.	.	.	.	.	Z	.
INX	Increment X-register	X <-- (Y) + 1	E8	2	N	.	.	.	.	.	Z	.
INY	Increment Y-register	Y <-- (Y) + 1	C8	2	N	.	.	.	.	.	Z	.
NOP	No Operation	Does Nothing	EA	2	.	.	.	.	.	.	.	.
PHA	Push A-register on stack	M(S) <-- (A), S <-- (S) − 1	48	3	.	.	.	.	.	.	.	.
PHP	Push P-register on stack	M(S) <-- (P), S <-- (S) − 1	08	3	.	.	.	.	.	.	.	.
PLA	Pull Stack to A-register	S <-- (S) + 1, A <-- (M(S))	68	4	N	.	.	.	.	.	Z	.
PLP	Pull Stack to P-register	S <-- (S) + 1, A <-- (M(S))	28	4	restored							
RTI	Return from Interrupt	Pull to P and PC	40	6	restored							
RTS	Return from Subroutine	Pull to PC	60	6	.	.	.	.	.	.	.	.
SEC	Set Carry	C <-- 1	38	2	.	.	.	.	.	.	.	C
SED	Set Decimal Mode	D <-- 1	F8	2	.	.	.	.	D	.	.	.
SEI	Set Interrupt Mask	I <-- 1	78	2	.	.	.	.	.	I	.	.
TAX	Transfer A to X	X <-- (A)	AA	2	N	.	.	.	.	.	Z	.
TAY	Transfer A to Y	Y <-- (A)	A8	2	N	.	.	.	.	.	Z	.
TSX	Transfer S to X	X <-- (S)	BA	2	N	.	.	.	.	.	Z	.
TXA	Transfer X to A	A <-- (X)	8A	2	N	.	.	.	.	.	Z	.
TXS	Transfer X to S	S <-- (X)	9A	2	.	.	.	.	.	.	.	.
TYA	Transfer Y to A	A <-- (Y)	98	2	N	.	.	.	.	.	Z	.

STATUS REGISTER

7	6	5	4	3	2	1	0
N	V	*	B	D	I	Z	C

Bit	Value = 0	Value = 1
N	Negative	Last result + ($00-$7F).	Last result − ($80−$FF).
V	Overflow	Last result no overflow, and after CLV.	Last result overflow.
*	Unused	Never.	Always.
B	Break		After BRK.
D	Decimal	After CLD (Binary mode).	After SED (Decimal mode).
I	Interrupt	After CLI (IRQ enabled).	After SEI (IRQ disabled).
Z	Zero	Last result non-zero.	Last result zero.
C	Carry	Last result no Carry, and after CLC.	Last result did carry, and after SEC.

6502 INSTRUCTIONS WITH OPCODE AND EXECUTION CYCLES

		Accumulator	Immediate	Zero Page	Absolute	Zero Page, X	Absolute, X	Zero Page, Y	Absolute, Y	(Zero Page, X)	(Zero Page), Y	
ADC	Add with Carry A <-- (A) + (M) + (C)	--	69 2	65 3	6D 4	75 4	7D *4	-- --	79 *4	61 6	71 *5	N V ... Z C
AND	Logical "And" A <-- (A) and (M)	--	29 2	25 3	2D 4	35 4	3D *4	-- --	39 *4	21 6	31 *5	N ... Z .
ASL	Shift Byte Left C <-- [7.....0] <-- 0	0A 2	--	06 5	0E 6	16 6	1E 7	--	--	--	--	N ... Z C
BIT	Test Bits in Memory Z <-- (A) and (M), N <-- (M) bit 7, V <-- (M) bit 6	--	-- 3	24 4	2C --	--	--	--	--	--	--	N V ... Z .
CMP	Compare with A-register N, Z, C <-- (A) − (M)	--	C9 2	C5 3	CD 4	D5 4	DD *4	-- --	D9 *4	C1 6	D1 *5	N ... Z C
CPX	Compare with X-register N, Z, C <-- (X) − (M)	--	E0 2	E4 3	EC 4	--	--	--	--	--	--	N ... Z C
CPY	Compare with Y-register N, Z, C <-- (Y) − (M)	--	C0 2	C4 3	CC 4	--	--	--	--	--	--	N ... Z C
DEC	Decrement Memory Byte M <-- (M) − 1	--	-- 5	C6 6	CE 6	D6 6	DE 7	--	--	--	--	N ... Z .
EOR	Exclusive-Or A <-- (A) eor (M)	--	49 2	45 3	4D 4	55 4	5D *4	-- --	59 *4	41 6	51 *5	N ... Z .
INC	Increment Memory Byte M <-- (M) + 1	--	-- 5	E6 6	EE 6	F6 6	FE 7	--	--	--	--	N ... Z .
LDA	Load A-register A <-- (M)	--	A9 2	A5 3	AD 4	B5 4	BD *4	-- --	B9 *4	A1 6	B1 *5	N ... Z .
LDX	Load X-register X <-- (M)	--	A2 2	A6 3	AE 4	--	--	B6 4	BE *4	--	--	N . Z .
LDY	Load Y-register Y <-- (M)	--	A0 2	A4 3	AC 4	B4 4	BC *4	--	--	--	--	N ... Z .
LSR	Shift Byte Right 0 --> [7.....0] --> C	4A 2	--	46 5	4E 6	56 6	5E 7	--	--	--	--	N ... Z C
ORA	Inclusive-Or A <-- (A) or (M)	--	09 2	05 3	0D 4	15 4	1D *4	-- --	19 *4	01 6	11 *5	N ... Z .
ROL	Roll Byte Left C <-- [7.....0] <-- C	2A 2	--	26 5	2E 6	36 6	3E 7	--	--	--	--	N ... Z C
ROR	Roll Byte Right C --> [7.....0] --> C	6A 2	--	66 5	6E 6	76 6	7E 7	--	--	--	--	N ... Z C
SBC	Subtract with Borrow A <-- (A)-(M) + (C)-1	--	E9 2	E5 3	ED 4	F5 4	FD *4	-- --	F9 *4	E1 6	F1 *5	N V ... Z C
STA	Store A-register M <-- (A)	--	-- 3	85 4	8D 4	95 5	9D --	-- --	99 5	81 6	91 6	. . ..
STX	Store X-register M <-- (X)	--	-- 3	86 4	8E 4	--	--	96 4	--	--	--	. . ..
STY	Store Y-register M <-- (Y)	--	-- 3	84 4	8C 4	94 --	--	--	--	--	--	. . ..

*Add one cycle if indexing causes a page boundary to be crossed.

INSTRUCTIONS WITH OPCODE AND UTION CYCLES

Branch if Plus	(Branch if N=0)	10	2 cycles if no branch;
Branch if Minus	(Branch if N=1)	30	
Branch if No Overflow	(Branch if V=0)	50	3 cycles if branch into same page;
Branch if Overflow	(Branch if V=1)	70	
Branch if Carry Clear	(Branch if C=0)	90	
Branch if Carry Set	(Branch if C=1)	B0	4 cycles if branch into different page.
Branch if Not Equal	(Branch if Z=0)	D0	
Branch if Equal	(Branch if Z=1)	F0	

MP INSTRUCTIONS

Unconditional Jump	Absolute	4C	3
Unconditional Jump	Indirect	6C	5
Jump to Subroutine		20	6

PLIED ADDRESS MODE INSTRUCTIONS

RK	Break (Set B=1, Generate IRQ Interrupt)		00	7	.	.	.	B	.	I	.	.
LC	Clear Carry	C <-- 0	18	2	.	.	.	.	.	.	.	C
LD	Clear Decimal Mode	D <-- 0	D8	2	.	.	.	.	D	.	.	.
LI	Clear Interrupt Mask	I <-- 0	58	2	.	.	.	.	.	I	.	.
LV	Clear Overflow Status	V <-- 0	B8	2	.	V	.	.	.	.	.	.
DEX	Decrement X-register	X <-- (X) − 1	CA	2	N	.	.	.	.	.	Z	.
DEY	Decrement Y-register	Y <-- (Y) − 1	88	2	N	.	.	.	.	.	Z	.
DEY	Decrement Y-register	Y <-- (Y) − 1	88	2	N	.	.	.	.	.	Z	.
INX	Increment X-register	X <-- (X) + 1	E8	2	N	.	.	.	.	.	Z	.
INY	Increment Y-register	Y <-- (Y) + 1	C8	2	N	.	.	.	.	.	Z	.
NOP	No Operation	Does Nothing	EA	2	.	.	.	.	.	.	.	.
PHA	Push A-register on stack	M(S) <-- (A), S <-- (S) − 1	48	3	.	.	.	.	.	.	.	.
PHP	Push P-register on stack	M(S) <-- (P), S <-- (S) − 1	08	3	.	.	.	.	.	.	.	.
PLA	Pull Stack to A-register	S <-- (S) + 1, A <-- (M(S))	68	4	N	.	.	.	.	.	Z	.
PLP	Pull Stack to P-register	S <-- (S) + 1, A <-- (M(S))	28	4	restored							
RTI	Return from Interrupt	Pull to P and PC	40	6	restored							
RTS	Return from Subroutine	Pull to PC	60	6	.	.	.	.	.	.	.	.
SEC	Set Carry	C <-- 1	38	2	.	.	.	.	.	.	.	C
SED	Set Decimal Mode	D <-- 1	F8	2	.	.	.	.	D	.	.	.
SEI	Set Interrupt Mask	I <-- 1	78	2	.	.	.	.	.	I	.	.
TAX	Transfer A to X	X <-- (A)	AA	2	N	.	.	.	.	.	Z	.
TAY	Transfer A to Y	Y <-- (A)	A8	2	N	.	.	.	.	.	Z	.
TSX	Transfer S to X	X <-- (S)	BA	2	N	.	.	.	.	.	Z	.
TXA	Transfer X to A	A <-- (X)	8A	2	N	.	.	.	.	.	Z	.
TXS	Transfer X to S	S <-- (X)	9A	2	.	.	.	.	.	.	.	.
TYA	Transfer Y to A	A <-- (Y)	98	2	N	.	.	.	.	.	Z	.

STATUS REGISTER

7	6	5	4	3	2	1	0
N	V	*	B	D	I	Z	C

Bit	Value = 0	Value = 1
N Negative	Last result + ($00-$7F).	Last result − ($80−$FF).
V Overflow	Last result no overflow, and after CLV.	Last result overflow.
* Unused	Never.	Always.
B Break		After BRK.
D Decimal	After CLD (Binary mode).	After SED (Decimal mode).
I Interrupt	After CLI (IRQ enabled).	After SEI (IRQ disabled).
Z Zero	Last result non-zero.	Last result zero.
C Carry	Last result no Carry, and after CLC.	Last result did carry, and after SEC.

6502 INSTRUCTIONS WITH OPCODE AND EXECUTION CYCLES

		Accumulator	Immediate	Zero Page	Absolute	Zero Page, X	Absolute, X	Zero Page, Y	Absolute, Y	(Zero Page, X)	(Zero Page), Y	
ADC	Add with Carry A <-- (A) + (M) + (C)	--	69 2	65 3	6D 4	75 4	7D *4	--	79 *4	61 6	71 *5	N V . . . . Z C
AND	Logical "And" A <-- (A) and (M)	--	29 2	25 3	2D 4	35 4	3D *4	--	39 *4	21 6	31 *5	N . . . . . Z .
ASL	Shift Byte Left C <-- [7 . . . . . 0] <-- 0	0A 2	--	06 5	0E 6	16 6	1E 7	--	--	--	--	N . . . . . Z C
BIT	Test Bits in Memory Z <-- (A) and (M), N <-- (M) bit 7, V <-- (M) bit 6	--	24 3	2C 4	--	--	--	--	--	--	--	N V . . . . Z .
CMP	Compare with A-register N, Z, C <-- (A) − (M)	--	C9 2	C5 3	CD 4	D5 4	DD *4	--	D9 *4	C1 6	D1 *5	N . . . . . Z C
CPX	Compare with X-register N, Z, C <-- (X) − (M)	--	E0 2	E4 3	EC 4	--	--	--	--	--	--	N . . . . . Z C
CPY	Compare with Y-register N, Z, C <-- (Y) − (M)	--	C0 2	C4 3	CC 4	--	--	--	--	--	--	N . . . . . Z C
DEC	Decrement Memory Byte M <-- (M) − 1	--	--	C6 5	CE 6	D6 6	DE 7	--	--	--	--	N . . . . . Z .
EOR	Exclusive-Or A <-- (A) eor (M)	--	49 2	45 3	4D 4	55 4	5D *4	--	59 *4	41 6	51 *5	N . . . . . Z .
INC	Increment Memory Byte M <-- (M) + 1	--	--	E6 5	EE 6	F6 6	FE 7	--	--	--	--	N . . . . . Z .
LDA	Load A-register A <-- (M)	--	A9 2	A5 3	AD 4	B5 4	BD *4	--	B9 *4	A1 6	B1 *5	N . . . . . Z .
LDX	Load X-register X <-- (M)	--	A2 2	A6 3	AE 4	--	--	B6 4	BE *4	--	--	N . . . . . Z .
LSR	Shift Byte Right 0 --> [7 . . . . . 0] --> C	4A 2	--	46 5	4E 6	56 6	5E 7	--	--	--	--	N . . . . . Z C
ORA	Inclusive-Or A <-- (A) or (M)	--	09 2	05 3	0D 4	15 4	1D *4	--	19 *4	01 6	11 *5	N . . . . . Z .
ROL	Roll Byte Left C <-- [7 . . . . . 0] <-- C	2A 2	--	26 5	2E 6	36 6	3E 7	--	--	--	--	N . . . . . Z C
ROR	Roll Byte Right C --> [7 . . . . . 0] --> C	6A 2	--	66 5	6E 6	76 6	7E 7	--	--	--	--	N . . . . . Z C
SBC	Subtract with Borrow A <-- (A)-(M) + (C)-1	--	E9 2	E5 3	ED 4	F5 4	FD *4	--	F9 *4	E1 6	F1 *5	N V . . . . Z C
STA	Store A-register M <-- (A)	--	--	85 3	8D 4	95 4	9D 5	--	99 5	81 6	91 6	. . . .
STX	Store X-register M <-- (X)	--	--	86 3	8E 4	--	--	96 4	--	--	--	. . . .
STY	Store Y-register M <-- (Y)	--	--	84 3	8C 4	94 4	--	--	--	--	--	. . . .

*Add one cycle if indexing causes a page boundary to be crossed.

S-C Software Corporation

2331 Gus Thomasson, Suite 125, P.O. Box 280300, Dallas, Texas 75228 (214) 324-2050

Motorola 6805 Version

The S-C Macro 6805 Cross Assembler is a complete macro assembler with co-resident program editor. It is written in 6502 assembly language for execution in the Apple II. It assembles standard 6805 mnemonics, as given in the Motorola M6805/M146805 Family Microcomputer/Microprocessor User's Manual.

The assembled object code may be directed either to Apple memory or to a DOS 3.3 Binary file. If you have an EPROM burner, the data can be burned into EPROMs.

The 6805 version's .DA directive outputs the high byte first, followed by the low byte, according to 6805 convention. (The 6502 version stores the low byte first.) The # and / can still be used to get only the low or high byte, however. All other directives are unchanged from the 6502 version.

When an instruction has more than one operand, there must be commas between the operands and no spaces. There must be at least one space between the operands and the opcode, and at least one space between the operands and the comments.

The BRSET and BRCLR (BRanch if bit SET or CLeaR) instructions have three operands. The first is the bit number, the second is the location to be tested, and the third is the destination of the branch. Examples:

```
START   BRSET 5,$30,END    branch to END if bit 5
                           of location $30 is set.
END     BRCLR BITNO,ZPAGE,START branch to START
                           if bit number BITNO

                           of ZPAGE is clear
```

(Note that the bit number may be a symbol.)

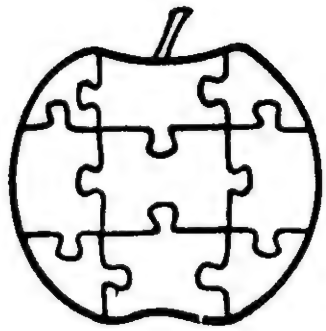
The BSET and BCLR (Bit SET and CLeaR) instructions have two operands: the bit number and the location to be tested. Examples:

```
BSET 7,FLAG
BCLR HIBIT,$43
```

The Indexed addressing modes must have a ,X after the address. A one byte offset is used if possible, otherwise two bytes are used. Notice that the 0,X and the ,X addressing modes are different. The 0,X uses two bytes (opcode and offset), while ,X uses only one (opcode only). They are functionally the same, however.

S-C Macro Assembler

"Makes assembly language programming on the Apple as easy as programming in BASIC."



S-C Software Corporation
2331 Gus Thomasson, Suite 125
P.O. Box 280300
Dallas, Texas 75228
(214) 324-2050

Chapter 1 -- INTRODUCTION

S-C Software Corporation is pleased to introduce the S-C Macro Assembler, the latest version of our most popular product. The S-C Assembler II Version 4.0 already has the reputation of being the easiest editor/assembler to learn, to remember, and to use...now the S-C Macro Assembler provides a new level of power and performance for the beginner and professional programmer alike.

The S-C Macro Assembler boasts 20 directives (pseudo-ops) and 29 commands, including a convenient EDIT command with 15 subcommands. COPY and REPLACE commands further simplify entry and modification of even the most complex programs.

The S-C Macro Assembler will operate in any Apple II or Apple II Plus with at least 32K RAM and one disk drive. Any additional memory or disk drives will be used as required. A Language Card version is also included.

A memory size of 48K allows source programs of over 24,000 bytes to be handled entirely within RAM. The Language Card version allows source programs of over 32,000 bytes. Much larger programs can be edited and assembled using the "INCLUDE" and "TARGET FILE" capabilities, up to the limit of on-line disk storage.

Programs can be edited, assembled, and tested entirely within the framework of the S-C Macro Assembler. The editor and assembler are co-resident, allowing rapid cycles of modification, re-assembly, and check-out. All DOS and Apple Monitor commands are active as well, providing a familiar interface to the standard Apple features.

The S-C Macro Assembler uses its own technique to store source files, but it also can read or write standard TEXT files. With this ability, you can EXEC in files from another assembler, use some other text editor to prepare files, keep a library of routines on disk to EXEC into any program, or use S-C Macro Assembler to prepare EXEC files for any purpose.

Already well-known for excellent support, S-C Software Corporation pledges to continue development of new features, and to help owners gain the maximum benefit from the S-C Macro Assembler. In addition to telephone consultation, a monthly newsletter is available by subscription (currently $15/year). The "Apple Assembly Line" covers items of interest to assembly language programmers at all levels, and has helped many to advance their programming skills.

New Features

Here is a brief summary of the new features the S-C Macro Assembler has that S-C Assembler II Version 4.0 did not. For more details on these new features, see the relevant chapters of this manual.

The highlights are of course macros, conditional assembly and the new commands EDIT, COPY and REPLACE. But they are not all!

COMMANDS

There are 10 new commands:

EDIT	Select a line, a range of lines, or a range of lines that contain a particular string. Edit the lines using some of the 15 convenient sub-commands.
TEXT	Write source program to disk, as a TEXT file. Write it with or without line numbers!
REPLACE	Global search and replace. Your search string can include wildcards; you can limit the search to a line, a range of lines, or search the entire program. The search can be made sensitive or insensitive to upper/lower case distinctions. And you can select Auto or Verify mode for replacement.
COPY	Copy one or more lines from one place to another in the source code. Rearrange your code as you please!
AUTO	Generate automatic line numbers after every carriage return. Allows ordinary TEXT files to be EXECed into S-C Macro Assembler! You still can use the Version 4.0 form of automatic line numbers. Now you have a choice!
MANUAL	Turn off automatic line numbering.
SYMBOLS	Print out the symbol table, in case you missed the first time.
MNTR	Enter the system monitor (just like CALL -151 in BASIC). Of course all the Monitor commands can be executed within S-C Macro Assembler, but if you really WANT to leave....
RST	Change the Autostart Monitor RESET vector to the specified address.
*	Send setup control strings to your printer.

There are also improvements in the older commands:

The spelling of commands is now checked. In older versions, only the first three characters were tested. The first three are still all that are necessary, but any additional letters you type must be correct. For example, LIS will list your program, and so will LIST. But, LISX will give a syntax error.

LIST and FIND now have the same syntax (in fact, they are processed by the same routine.) They may now specify either a line range, a search string, or both. The search string now requires a delimiter.

Line ranges in the LIST, FIND, COPY, EDIT, and DELETE commands may be written with a leading or trailing comma (as in Applesoft):

```
LIST ,2500      List from beginning through 2500.
LIST 2500,      List from 2500 through end.
```

The NEW command now restarts the automatic line numbering at 1000, rather than continuing from the last line number you entered.

The SLOW and FAST commands no longer use the Monitor output hooks at $36 and $37.

To leave the Macro Assembler, type FP or INT. You no longer have to also type PR#0.

After using the PR#slot command to run your printer, use PR#0 to turn it off. FAST won't do it anymore.

DIRECTIVES

There are 7 new directives:

.MA and .EM	For macro definition.
.DO expr	Start conditional block.
.ELSE	Toggle condition flag.
.FIN	End conditional block.
.TI num,title	Title and number each page of the assembly listing.
.AT string	Like .AS, but the last character has the high-bit set opposite from the rest.

The .DA directive may now have a list of expressions.

The .EQ directive may now be used with local labels.

The .LIST directive has new options to control listing of macro expansions.

SOURCE ENTRY

Control-O (Override) will allow any control character to be typed into a source line in the normal input mode or in edit mode. The control character will appear in inverse video.

The editor no longer double spaces after each line is entered.

The escape-L comment line produces one less dash, so that the line lists on the screen without a blank line after it.

Operand expressions can now include * and / as operators, as well as + and -. The relational operators (<, =, and >) may also be used.

The tab routine has been changed to include up to five tab stops. The stop values are kept in a user-modifiable list starting at $1010. These are the actual column numbers (not 3 less, as in version 4.0). You may use any values up to column 248.

The tab character (control-I, $89) is kept at $100F now, so you can change it if you like some other character better.

Any sequence of the same character repeated 4 or more times in the source code is replaced by a token $C0, the character code, and the repeat count. (multiple blanks are still replaced by a single byte between $80 and $BF.) This reduces both the memory requirements and disk file size for your source programs.

If you want to shrink your source file a little, and if you have been using the Escape-L to generate comment lines that have all those dashes in them, type "EDIT", RETURN and hold down the CTRL, L, and REPEAT keys until the entire program has been scanned. Type MEM before you do it, and after it is finished; you will probably notice a significant saving!

A parameter at location $1017 allows the extra compression to be turned on or off. If the contents of $1017 is $04, compression is on. If it is $FF, compression is off. You can experiment with this parameter to see what effect it has on program size.

ASSEMBLY

Older versions of the assembler terminated assembly after finding one error. The S-C Macro Assembler keeps going, but rings the bell and prints an error message, so you know about it. If any errors are found during pass one, assembly terminates before doing pass two. At the end of assembly, the number of errors found is printed.

Typing the RETURN key during assembly will abort the assembly (even if the listing has been turned off with .LIST OFF directive).

MEMORY USAGE

All page zero variables used by the assembler have been concentrated, so $00 through $1F are completely free for the user.

The standard version of the S-C Macro Assembler now occupies $1000 through $31FF. The symbol table starts at $3200 and grows upward; the source code still starts at $9600 and grows downward.

The Language Card version fills the 16K RAM card from $D000 through about $F300. The symbol table begins at $1000 rather than $3200.

There are no variables within the body of the assembler. The Language Card version could be burned into ROM and placed on a firmware card, if you so desire.

Chapter 2 -- TUTORIAL

In this chapter, we'll go step-by-step through the process of writing a small program with the S-C Macro Assembler.

First enter S-C Macro Assembler from DOS by typing "BRUN S-C.ASM.MACRO" or by booting the disk.

Then type in the following short program:

```
1000 TONE   LDA $C030
1010 LOOP   DEY
1020        BNE LOOP
1030        JMP TONE
```

Now type "LIST" to see the program as the computer has it. The display should look like this:

```
:LIST
 1000 TONE   LDA $C030
 1010 LOOP   DEY
 1020        BNE LOOP
 1030        JMP TONE
:
```

Description of the Source Program:

The listing above is called a source program. It is the text form of an assembly language program. Later we will go through the steps necessary to convert it to executable form, but for now let's observe what the source form looks like.

The first column contains line numbers. These are always 4-digit numbers. Assembler line numbers work just like BASIC line numbers for editing, inserting and deleting lines, but have nothing to do with the flow of control (no GOTO linenumber.)

The second column contains labels. These are used instead of line numbers for controlling the program flow. They also can act like BASIC variables. In our example, the labels are TONE and LOOP.

The third column contains opcodes (OPeration codes). These are either standard 6502 instructions, SWEET-16 instructions, macro calls, or special "directives" to the S-C Macro Assembler. In our example, all the opcodes are 6502 instructions (LDA, DEY, BNE, and JMP).

The fourth column contains operands. The opcode tells the computer what to do; the operand tells what to do it to. The operand can be a number, a label, or an arithmetic expression. Sometimes the opcode does not use an operand, as in the DEY above. Others use a more complicated format. The operand on the first line above, "$C030", is a hexadecimal number. The operands on the BNE and JMP lines are labels.

Saving a Source Program on Disk:

To save the program to a disk, type "SAVE NOISY". This is a standard DOS SAVE command, just like you would use in BASIC. Note that S-C Macro Assembler source files are type "I" files. DOS thinks they are Integer BASIC programs, but they will not run as they are. (DOS is fooled, but Integer BASIC would not be fooled at all.) NOTE: You do NOT need to have Integer BASIC in your machine to use the S-C Macro Assembler.

To clear memory for a new program type "NEW". To reload the program from disk type "LOAD NOISY".

Assembling a Source Program:

A program must be assembled into binary form before it can be executed. The command to assemble a program is "ASM". Try it now....

Our program is now assembled into memory starting at address $0800. The display should look like this:

```
:ASM

0800- AD 30 C0 1000 TONE   LDA $C030
0803- 88       1010 LOOP   DEY
0804- D0 FD    1020        BNE LOOP
0806- 4C 00 08 1030        JMP TONE

SYMBOL TABLE

0803- LOOP
0800- TONE
:
```

Notice that two more columns have been added to the left of those we saw when we typed "LIST". The first new column contains the memory addresses (in hexadecimal) into which the program assembled. The second column has one, two, or three hex numbers (two digits each) which are the contents of the memory locations.

The Symbol Table is a list of all the labels and the values assigned to them.

The program is now in memory in two forms. The source program is there, right beneath DOS. The executable form, called the "object" program, is in memory from $0800 through $0808.

Executing the Object Program:

To run the program, type "MGO TONE". Do you hear the tone coming from your Apple speaker? It is being produced by continually toggling the position of the speaker coil by addressing $C030, at about 800 toggles per second. This makes a tone of about 400 Hertz.

As soon as you get tired of it, hit the RESET key to stop the noise. Note that you can run a program from the assembler by MGO-ing to a label, and that the RESET key reenters the assembler. (If you have the Autostart ROM, that is. If you don't, you will get the "*" prompt: type 3D0G to reenter the assembler).

Now we have walked through the entry, assembly, and execution of a very small program. The same steps would work for a large program, but there are many other features built-in to the S-C Macro Assembler which can make programming in assembly language even easier than programming in BASIC.

Modifying a Source Program:

To get the flavor of some more of the features of the S-C Macro Assembler, let's modify the TONE program a little.

It would be nice if the TONE program would stop gracefully without hitting the RESET key. I like to set it up to quit when any key is pressed on the keyboard. I do it like this:

```
1030        LDA $C000     LOOK AT KEYBOARD
1040        BPL TONE      NO KEY PRESSED YET
1050        STA $C010     CLEAR KEYBOARD STROBE
1060        RTS           RETURN
```

Note that a new column has been added: the comments column. The operand column ends with a blank; any useful comments to help you understand next week what you did today can be written after that blank.

If you type in those four lines, and then type the LIST command, you will see that they are now part of your source program. Line 1030 has been replaced, since you typed a new line 1030.

Now type ASM, to create the new version of your program in executable (object) form. And execute it, by typing MGO TONE. This time you can stop the tone by pressing any key.

Easier Entry of Source Programs:

Now let's try an easier way to enter source lines. First save the latest version of your TONE program on a disk by typing SAVE NOISY again. Then type "NEW" to erase the source program from memory. (It is still on the disk.)

Now hold down the CTRL key ("CTRL" means "control"), and type the letter I. We call that typing "control-I". Look at the screen. You will see that the Apple printed "1000 ", and the cursor is blinking after that. Type control-I again, and you will see the cursor move over about 7 character positions.

Whenever you type control-I at the beginning of a line, the S-C Macro Assembler will automatically generate the next line number for you. Usually this will be ten higher than the last line number you entered or deleted. (The increment is settable to whatever interval you like.)

Whenever you type control-I inside a line (beyond the beginning), the cursor will move to the next tab stop. Play with this a while, and you will find that the tab stops line up nicely with the source program format.

Why don't you try typing in the TONE program again, with a few more comments for good measure? This time use control-I for the line numbers and tabbing. Start by typing NEW, so we know that there are no stray line numbers left from some previous work. I am going to help you, showing control-I with the symbol "^I". Now type:

```
:^ITONE^ILDA $C030^ITOGGLE THE APPLE'S SPEAKER
:^ILOOP^IDEY^I^IDELAY FOR ABOUT 1250 MICROSECONDS
:^I^IBNE LOOP
:^I^ILDA $C000^ILOOK AT KEYBOARD
:^I^IBPL TONE^INO KEY PRESSED YET
:^I^ISTA $C010^ICLEAR KEYBOARD STROBE
:^I^IRTS^I^IRETURN
:LIST
```

Type ASM again, and NGO TONE. The program should work just like it did the first time. Stop it by pressing any key.

With this brief introduction, you should now be ready to dive into the following chapters. As you study each new command or feature, experiment with it until you really understand what is happening. Look up some of the books mentioned in the Reference Bibliography for help in understanding the 6502 language. If you need some personal help, call us at (214) 324-2050.

Source programs are entered a line at a time, with a line number identifying each line. The line numbers may run from 0 through 9999. The automatic line numbering and the RENUMBER command use numbers from 1000 through 9999 to keep the columns straight, but this is not necessary if you prefer lower numbers. Source program lines are kept sorted in line-number order; the numbers are used for editing purposes, just as in BASIC.

A blank must always follow the line number. After the blank there are four fields of information: the label, opcode, operand, and comment fields. Adjacent fields must be separated by at least one blank. Lines may be up to 248 characters long.

Automatic Line Numbering:

Although you may type the line number yourself, in the same way as in either Applesoft or Integer BASIC, the S-C Macro Assembler includes two convenient and powerful means for automatic line number generation.

The first method is really semi-automatic, because you do have to type a control-I to get the next line number. Any time the cursor is at the beginning of a line (right after the prompting colon), typing a control-I will cause the next line number to be generated. Immediately after loading the Macro Assembler or typing "NEW", the "next line number" will be 1000. The number will be displayed as four digits and a trailing blank. The cursor will be in position for the first character of a label, or the asterisk denoting a comment line. If you type the control-I in any other position than the beginning of a line, it will cause a tab to the next tab stop.

The second method is invoked by typing the AUTO command, with or without a starting line number. In the AUTO mode, the next line number is automatically generated at the beginning of every line. If you don't want to use the line number, or want one out of sequence, you can backspace up to the prompt and type a command or line number. The AUTO mode is terminated by the MANUAL command, by hitting RESET, or by any error message.

The "next number" is always the value of the previous line number plus the current "increment". The standard increment i 10, but you can change it to any reasonable value with the INCREMENT command.

Built-In Tab Stops:

Although the opcode, operand, and comment fields are not required to begin in any particular column, it is neater to align them. Therefore tab stops are included in S-C Macro Assembler at columns 14, 18, 27, and 32.

The standard tab stops allow a label field of six characters, an opcode field of three characters, and a comment field beginning in column 41 of the assembly listing.

Control-I is the standard ASCII tab character, and is used by the S-C Macro Assembler. Normally control-I will generate enough spaces to move the cursor to the next tab stop. If control-I is typed at the beginning of a line, the next line number and one space will be generated. If you are already past or at the last tab stop, control-I will generate a single space.

Some printer interface cards with firmware drivers use control-I for setting various modes. If you wish to change the tab character, you may do so. It is stored at location $100F. An alternative is to change the printer interface control character, which is usually stored at $06F8+slot#.

Space has been reserved inside S-C Macro Assembler for a total of five tab stops. They are stored in locations $1010 through $1014, as column numbers. You may change them if you wish. If you want fewer than five tab stops, set the remaining ones to zero.

Label Field:

The label field may be left blank, or may contain a label. There are three types of labels used in S-C Macro Assembler: normal labels, local labels, and private labels. The first character of the label must be in the second column after the line number.

```
1000 START.HERE    (normal label)
1010 .23           (local label)
1020 :12           (private label)
```

Normal Labels: Used to name places in your program to which you will branch, as well as constants and variables. Normal labels may be up to 32 characters long. The first character must be a letter; subsequent characters may be letters, digits, or the period character (".") . The period is useful for making long labels readable. For example, a subroutine to extract the next character from a buffer might be named "GET.NEXT.CHAR.FROM.BUFFER".

The standard tab stops assume your labels will be six or less characters long. However, since the assembler is relatively free-format, you may type any length label followed by a blank and the opcode, operand, and comment fields. Or, if you wish, you may type the long label on a line all by itself. In this form, the label is assigned the current value of the location counter, just as if you had appended ".EQ *" to the line.


```
1000 *   SAMPLE PROGRAM WITH NORMAL LABE.
1010 *
1020 SOURCE.LINE.POINTER .EQ $13 (WITH $14)
1030 CHAR.POINTER        .EQ $12
1040 *
1050 READ.NEXT.CHARACTER.FROM.LINE
1060         LDY CHAR.POINTER
1070         LDA (SOURCE.LINE.POINTER),Y
1080         INC CHAR.POINTER
1090         RTS
```

Local Labels: Used to name branch points within a module. The main purpose for local labels is to make programs more readable by reducing the number of label names you must invent. As a side effect, local labels save considerable space in the symbol table during assembly; they only require two bytes each. The use of local labels also encourages structured programming habits.

Local labels have a period as the first character, followed by one or two digits. Any label from ".0" through ".99" may be used. (Please note that these are label names, not decimal fractions. Consequently, the label ".1" is treated as exactly equivalent to the label ".01"; in fact, it will be listed in the symbol table listing as ".01".)

A local label is defined internally relative to the normal label which comes before it in the source program. (There must be one before it, or you will get an error message.) The value must be no more than 255 greater than the value of the associated normal label.

Since each set of local labels is associated with a particular normal label, you may re-use the same local labels as often as you wish.

Here is an example of three little routines in the same source program, using normal and local labels.

```
1000 PRINT.MESSAGE
1010         PHA              SAVE A-REGISTER
1020 .1      JSR PRINT.CHARACTER
1030         INY
1040         LDA MESSAGES,Y
1050         BNE .1           =0 FOR END OF MESSAGE
1060         PLA              RESTORE A-REGISTER
1070         RTS
1080 *
1090 GET.NEXT.CHARACTER
1100         LDY CHAR.POINTER
1110         LDA INPUT.BUFFER,Y
1120         CMP #RETURN
1130         BEQ .1           END OF LINE
1140         INC CHAR.POINTER
1150 .1      RTS
```

```
1160 *
1170 GET.NEXT.NONBLANK.CHAR
1180 .1     JSR GET.NEXT.CHARACTER
1190        BEQ .2          END OF LINE
1200        CMP #BLANK
1210        BEQ .1
1220 .2     RTS
```

Private Labels: Used primarily within macros as branch points. Private labels are maintained in a separate symbol table, and hence do not interfere with either normal labels or local labels. Each private label is associated with a particular invocation of a macro, so that the assembler treats the recurrence of the same label number as a unique label. Private labels are discussed in more depth in the chapter on MACROS.

Opcode Field:

The opcode field contains a machine language or SWEET-16 operation code, a macro name, or an assembler directive. If you are using the tab stops, the opcode field normally starts in column 14. However, opcodes may begin in any column after at least one blank from a label or at least two blanks from a line number.

The S-C Macro Assembler uses the standard 6502 instruction mnemonics as defined by MOS Technology, and the standard SWEET-16 mnemonics as defined by author Steve Wozniak. The 6502 opcodes, SWEET-16, macros, and assembler directives are all discussed later in this manual.

Operand Field:

The operand field usually contains an operand expression. Some of the 6502 instructions have no written operand, such as NOP, BRK, DEX, and others. Four of the 6502 instructions (ROL, ROR, ASL, and LSR) may be used either with or without a written operand. If no operand is written with these four instructions, you must type at least two blanks before a comment.

Comment Field:

Comments are separated from the operand field by at least one blank. For your convenience, a tab stop is set at column 26. In the assembly listing, tabbed comments will begin in column 41, which is the beginning of the next line on your Apple screen.

Comment Lines:

Full lines of comments may be entered by typing an asterisk (*) in the first column of the label field. This kind of comment is useful in separating various routines from each other, and labelling their contents. It is analogous to the REM statement in BASIC.

Lines which are completely blank are also treated as comments.

Escape-L:

A special comment line is built-in to the S-C Macro Assembler. If you are at the beginning of the label field (where a comment line could begin), typing the ESC key and then the letter "L" will generate the built-in comment line. The built-in comment line is a line of dashes which just fill one line on the screen. It is very commonly used to set off blocks of comments.

If a comment line of dashes is not your favorite, you may change the repeated character. The ASCII code of the character is kept at $1015. It is currently $AD, which is ASCII for "-".

If you type Escape-L at the beginning of a line (before a line number), it has a different meaning. In this case it will cause the first six characters of the line on the screen to be changed to " LOAD ". Then the rest of the line will be read from the screen, and issued as a LOAD command. With this feature you can LOAD a file simply by typing CATALOG, ESC-II. .IL.

Cursor Controls:

The standard Apple II screen editing tools are supported by S-C Macro Assembler. You can edit lines of assembly language source in just the same way that you edit lines in your Integer BASIC or Applesoft program.

Whether or not you have the Autostart ROM, you may use the new Apple standard cursor movement controls: escape-I, -J, -K, and -M. The older escape-A through escape-F and escape-@ are also supported by S-C Macro Assembler.

Chapter 4 -- COMMANDS

You will use three types of commands in S-C Macro Assembler: Assembler Commands, DOS Commands, and Monitor Commands. The Assembler Commands allow you to edit, assemble, and execute your assembly language programs. The Apple Monitor and DOS commands are also recognized, although they are not all of use from within the S-C Macro Assembler. Commands are typed immediately after the prompt symbol, which is a colon (:).

ASSEMBLER COMMANDS

There are 29 commands recognized by the S-C Macro Assembler. All Assembler Commands may be abbreviated to the first three letters if you so desire. As many characters of the command name that you do type are checked for spelling.

(Two DOS Commands, LOAD and SAVE, are used so frequently that they might be thought of as Assembler commands. However, they are DOS Commands, and as such cannot be abbreviated to the first three letters.)

The 29 Assembler Commands can be conveniently grouped into source commands, editing commands, listing control commands, object commands, and miscellaneous commands.

Group	Commands
Source	NEW, LOAD, SAVE, TEXT, HIDE, MERGE, and RESTORE
Editing	EDIT, COPY, LIST, FIND, REPLACE, DELETE, and RENUMBER
Listing	SLOW, FAST, PRT, and "
Object	ASM, MGO, VAL, and SYMBOLS
Miscellaneous	AUTO, MANUAL, INCREMENT, MEMORY, RST, MNTR, and USR

Source Commands

NEW, LOAD, SAVE, TEXT, HIDE, MERGE, RESTORE

Source Commands are used to erase the current program from memory. load a program from disk or tape, save a program on disk or tape, and append a program from disk or tape.

NEW Command:

```
:NEW
```

Deletes the current source program from memory and restarts S-C Macro Assembler. Clears the screen, writes "S-C MACRO ASSEMBLER II V1.0" on the top line. restarts the automatic line numbering at 1000 and waits for you to type a source line or another command.

LOAD Command:

```
:LOAD
:LOAD filename
```

Deletes the current source program (unless it is "hidden" with the HIDE command), and then reads in a new one from cassette tape. It works exactly the same as the LOAD command in Integer BASIC or Applesoft.

If you type a filename after the LOAD command, it is intercepted by DOS and a source program is loaded from disk instead of tape.

```
:LOAD              (load from tape)
:LOAD BANANA       (load disk file named "BANANA")
```

SAVE Command:

```
:SAVE
:SAVE filename
```

Writes the source program currently in memory to cassette tape. It works exactly the same as the SAVE command in Integer BASIC or Applesoft.

If you type a filename after the SAVE command, it will be intercepted by DOS to write the source program on disk rather than tape. It will appear in the disk directory as a type "I" file.

```
:SAVE              (save on tape)
:SAVE BANANA       (save on disk file "BANANA")
```


TEXT Command:

```
:TE    filename
:TEXT$ filename
:TEXT/ filename
```

Saves the source program to disk as a DOS text file, so it can be EXECed into this or another assembler. or edited with another text editor. There are three forms of this command:

TEXT writes the lines with no line number. This is very handy for building EXEC files for use with DOS. BASIC, or any general use. They can be read back into the S-C Macro Assembler by turning the AUTO line numbers on (see AUTO command), and using the EXEC command.

TEXT$ writes them with line numbers, exactly as they list on the screen. These files can be EXECed into Applesoft, or back into the S-C Macro Assembler.

TEXT/ writes the lines with a control-I in place of the line number. You can keep disk files of often-used routines that can be EXECed into a program wherever they are needed. The control-I at the beginning of each line will cause a line number to be generated when the line is read by the S-C Macro Assembler.

```
:TEXT ROUTINE   (writes the current source
                 program on a text file
                 named ROUTINE, with no line numbers)

:TEXT$ ROUTINE  (writes the current source
                 program on a text file
                 named ROUTINE, with line numbers.)

:TEXT/ ROUTINE  (writes the current source
                 program on a text file
                 named ROUTINE, with control-I's
                 in place of line numbers.)
```

HIDE and MERGE Commands:

```
:HIDE
:MERGE
```

These two commands, used together with the LOAD command, allow you to join a program from disk or tape to a program that is already in memory. To remind you that you are HIDE-ing, the prompt symbol changes from ":" to "H:". After HIDE-ing a program, you can LOAD another one from disk or tape with the LOAD command. Then you type MERGE to join the two programs together.

After this sequence of commands, the program which was already in memory will follow after the program just LOADed. If the line numbers are not already as you wish them to be, you should use RENUMBER to assign new ones.

For example, suppose that we have two source programs on the disk named "SRCONE" and "SRCTWO". We want to join them together so that "SRCONE" precedes "SRCTWO". Here are the two programs:

```
:LOAD SRCONE
:LIST

1000 *   PROGRAM NUMBER ONE
1010 MAIN    JSR SUBROUTINE
1020         RTS

:LOAD SRCTWO
:LIST

1000 *
1010 SUBROUTINE
1020         LDA BLAH.BLAH.BLAH
1030         STA SOMEWHERE
1040         RTS
```


Now let's HIDE the source of SRCTWO, LOAD in the source from SRCONE, and MERGE them together.

```
:HIDE
H:LIST          (Note that nothing lists, because the
H:LOAD SRCONE    source has been hidden.)
H:LIST

1000 *   PROGRAM NUMBER ONE
1010 MAIN    JSR SUBROUTINE
1020         RTS

H:MERGE

:LIST

1000 *   PROGRAM NUMBER ONE
1010 MAIN    JSR SUBROUTINE
1020         RTS
1000 *   SUBROUTINE TO DO SOMETHING
1010 SUBROUTINE
1020         LDA BLAH.BLAH.BLAH
1030         STA SOMEWHERE
1040         RTS
```

You can see that the two programs are now both in memory, but the line numbers are not in sequence. RENUMBER will fix the line numbers.

```
:RENUMBER
:LIST

1000 *   PROGRAM NUMBER ONE
1010 MAIN    JSR SUBROUTINE
1020         RTS
1030 *   SUBROUTINE TO DO SOMETHING
1040 SUBROUTINE
1050         LDA BLAH.BLAH.BLAH
1060         STA SOMEWHERE
1070         RTS
```

RESTORE Command: | :RESTORE |

Restores the root source program if an assembly is aborted while inside an "included" module.

The "root source program" is the source program that is in memory at the time you issue the ASM command. If this source program uses the .IN directive to include additional source files, it is possible that assembly might be aborted while the "root" program is "hidden". An assembly may be aborted either manually by typing a RETURN key while the assembly is in progress, or automatically due to an error in the source program.

If the assembly is aborted during the time that the root program is hidden, the prompt character changes from ":" to "I:". The RESTORE command will reset the memory pointers so that the included file is released, and the root program is no longer hidden. The prompt character will change back to ":".

You do not have to use the RESTORE command after an aborted assembly unless you wish to get back to the root source program for editing purposes. If you type the ASM command, the assembler automatically restores before starting the assembly.

If an assembly aborts due to an error in a source line, you may correct the source line. SAVE the module on the appropriate file, and type ASM to restart the assembly.

Editing Commands | EDIT, COPY, LIST, FIND, REPLACE, DELETE, RENUMBER

The editor in S-C Macro Assembler combines the Apple screen editing features with a BASIC-like line editor. Source programs are entered and edited in almost exactly the same way you would enter and edit an Integer BASIC or Applesoft program.

Editing commands allow you to list your source program, delete lines, search for lines, replace portions of selected lines, renumber lines, and copy blocks of lines from one location to another. There is also a powerful EDIT command, similar to Neil Konzen's Program Line Editor for BASIC.

Range Parameters:

Most editing commands (LIST, FIND, EDIT, REPLACE, AND DELETE) can use range parameters to operate on just part of the program. A range parameter may be written with one or two line numbers, or in most cases it may be omitted. If there are two line numbers, separate them with a comma. If there is only one line number, it may stand alone, or with a comma; the comma may precede or follow the line number. Each of these five possible arrangements has a specific meaning:

(no number)	Specifies the entire source program.
,	Specifies the entire source program.
#	Specifies line number #.
,#	Specifies lines from the beginning of the source program through #.
#,	Specifies lines from # through the end of the source program.
#1,#2	Specifies lines from #1 through #2.

Here are some specific examples:

:LIST	(lists all lines)
:LIST 2000	(lists line 2000 only)
:DEL 2000,3000	(deletes lines from 2000-3000)
:EDIT 2000,	(edits all lines from 2000 through the end of the program)
:FIND ,2000	(finds all lines from the beginning of the program through line 2000)

You can also use a period (.) to mean "the last line entered". The period, or "dot", is always defined as the number of the last line entered into or deleted from the source program.

String Parameters:

Some commands (LIST, FIND, EDIT, AND REPLACE) can also use a search string parameter to operate only on lines containing that string. The search string is of the form dstringd, where d is a delimiter of your choice. The delimiter can be any printing character that does not occur in your search string, except comma (,), period (.), or a digit (0-9). Some examples:

```
:LIST /COUT/     (lists all lines containing COUT)
:EDIT "/DEST"    (edits all lines containing /DEST)
```

You can use a wildcard character in search strings if you want to operate on all lines containing partial matches to your search string. The standard wildcard character is control-W. You have to first type a control-O, and then a control-W. The control-O character is an override to allow the insertion of control characters in commands and source lines. The control-W will appear on the screen in inverse video. For example:

```
:FIND ?ASWTA?     (imagine with me that the "W"
                   is a control-W)
 1100        LDA AS.DATA
 1120        LDA AS.DATA
 1200        STA BASKETA
```

LIST Command:

```
:LIST
:LIST range
:LIST dstringd
:LIST dstringd range
```

FIND Command:

```
:FIND
:FIND range
:FIND dstringd
:FIND dstringd range
```

Actually, FIND is just an alternate name for the LIST command. Many people find it more natural to use LIST with line number ranges and FIND with a search string, but either command will work with either parameter (or both parameters!).

Both FIND and LIST list a single line, a range of lines, or an entire source program. If you specify a search string, only those lines which match the string will be listed.

While a program or range of lines is listing, you can momentarily stop the listing by hitting the space bar. Tapping the space bar again will restart the listing. You can abort the listing by hitting the RETURN key. The SLOW and FAST commands allow you to control the listing speed. If you list a single line, it is displayed on the screen in a position which makes it easy to edit using the Apple screen editing tools.

```
:LIST              (list entire program)
:LIST 1230         (list only line 1230)
:LIST 1230,2890    (list lines 1230 through 2890)
:LIST 1230,        (list all lines from 1230 through end)
:LIST ,1230        (list all lines from beginning through
                       1230)
:FIND /ASCII/      (lists all lines containing the
                       string "ASCII")
:FIND "BI",1200    (lists all lines up through 1200 that
                       contain the string "BI")
```

EDIT Command:

```
:EDI
:EDI range
:EDIT dstringd
:EDIT dstringd range
```

Allows very easy editing of program lines. Since this command is typed so frequently, there is a short form: instead of typing "EDIT", you can just type control-E. The characters "EDIT " will magically appear on the screen; you fill in the line number, and proceed to edit the line.

If you specify no range or string, the whole source program, one line after another, will be presented for editing. If you specify a range, those lines in the range will be presented for editing. If you specify a search string, only those lines matching the string will be presented.

EDIT presents a line for editing by displaying that line, clearing from the end of that line to the bottom of the screen, and placing the cursor at the beginning of the label field. You can proceed to edit with the following commands:

control-B	Move cursor back to beginning of the label field.
control-D	Delete character under cursor.
control-Fx	Move cursor to next occurence of "x" in line (if any). You may type any character you wish for "x".
control-H	(left arrow) Move cursor left.
control-I	Begin insertion mode; characters will be inserted until another control character is typed.
control-L	Store current edited line and start editing the next line.
control-M	(RETURN) Store the edited line.
control-N	Move cursor to end of line.
control-O	Begin insertion mode, but allow next character typed to be inserted even if it is a control-character.
control-Q	Finish edit mode, chopping off all characters from cursor to end of line.
control-R	Restore the original line without leaving edit-mode.
control-T	Move cursor to next tab stop.
control-U	(right arrow) Move cursor right.
control-X	Abort the EDIT command.
control-@	Erase from cursor to end of line without leaving edit-mode.

REPLACE Command:

```
:REPLACE dstringdstri
:REPLACE dstringdstringd range
:REPLACE dstringdstringd options
:REPLACE dstringdstringd range options
```

Searches for and replaces character strings in your source code. REPLACE operates on all fields, from the first character in the label field through the end or each line. It can be global (search the entire program), or it can be made local by using range parameters to restrict which lines are searched.

When REPLACE finds your search string in a line it will print that line, with the matching string shown in inverse video. The program will then ask "REPLACE?", and wait for you to type "Y", "N", or some other character. If you type "Y" the corrected line will be listed, then the search will continue. If you type "N" it will simply continue searching. If you type some other character, the REPLACE command will terminate.

There are two possible options which may be selected by appending the letters "A" or "U" to the command line. A letter "A" on the end of the command line causes automatic operation, without the prompting at each replacement. A letter "U" means ignore any difference between upper and lower case letters.

It is possible to replace more than one matching string in the same source line.

```
:REP /CONT/GO.ON/          (change a label name)

1130        BNE CONT
REPLACE?  Y                          (The underlined
 1130        BNE GO.ON               characters appear
1210 *  NOW WE CAN CONTINUE          in inverse video.)
REPLACE?  N
1360 CONT   JSR BYTEIN
REPLACE?  Y
 1360 GO.ON  JSR BYTEIN
:
```

You may use wildcard characters in the search string. The entire matching string will be replaced with the replacement string. Do not put any wildcard characters in the replacement string.

DELETE Command:

```
:DELETE range
```

Deletes a line or range of lines from your source program, just as in BASIC. Another way to delete a single line is to type its line number followed immediately by a RETURN, or by a space and RETURN.

(Warning: DELETE followed by a file name is a DOS command, and will delete a file from your disk!)

DELETE must be followed by a range parameter and cannot have a search string parameter.

```
:DEL              (doesn't work)
*** SYNTAX ERROR
:DEL 1230         (delete only line 1230)
:DEL 1230,2890    (delete lines 1230 through 2890)
:DEL 1230,        (delete all lines from 1230 through end)
:DEL ,1230        (delete all lines from beginning through
                   1230)
:DEL ,            (delete entire program!)
```


RENUMBER Command:

```
:RENUMBER
:RENUMBER base
:RENUMBER  se,inc
:RENUMBER base,inc,start
```

Renumbers all or part of the lines in your source program with the specified starting line number and increment. There are three optional parameters for specifying the line number to assign the first renumbered line (base), the increment, and the place in your program to begin renumbering (start). There are four possible forms of the command:

```
:REN              Renumber the whole source program:
                  BASE=1000, INC =10. START=0

:REN #            Renumber the whole source program:
                  BASE=#, INC=10. START=0

:REN #1,#2        Renumber the whole source program:
                  BASE=#1. INC=#2. START=0

:REN #1.#2,#3 Renumber from line #3 through the end:
                  BASE=#1. INC=#2, START=#3
```

The last form above is useful for opening up a "hole" in the line numbers for entering a new section of code.

```
:LIST
 1000 *   LITTLE RENUMBER EXAMPLE
 1005 SAMPLE LDA $35
 1006        STA $37
 1010        RTS

:RENUMBER
:LIST
 1000 *   LITTLE RENUMBER EXAMPLE
 1010 SAMPLE LDA $35
 1020        STA $37
 1030        RTS

:RENUMBER 100
:LIST
 0100 *   LITTLE RENUMBER EXAMPLE
 0110 SAMPLE LDA $35
 0120        STA $37
 0130        RTS

:RENUMBER 2000,4
:LIST
 2000 *   LITTLE RENUMBER EXAMPLE
 2004 SAMPLE LDA $35
 2008        STA $37
 2012        RTS

:RENUMBER 3000,10,2008
:LIST
 2000 *   LITTLE RENUMBER EXAMPLE
 2004 SAMPLE LDA $35
 3000        STA $37
 3010        RTS
```

COPY Command:

```
:COPY range,target
```

Copies a range of lines from one place in the program to another. A copy of all the lines in the range specified is placed just before the target line.

If the target line does not exist, the range will be copied where the target line should have been. If the target is line 9999, and there is no line 9999, the copied lines will be placed at the end of the source program.

COPY does not delete the original section or renumber the copy, so this command should be followed immediately by a RENUMBER command.

```
:LIST
 1000 *   LITTLE COPY EXAMPLE
 1005 SAMPLE LDA $35
 1006        STA $37
 1010        RTS

:COPY 1005,1006.9999
:LIST
 1000 *   LITTLE COPY EXAMPLE
 1005 SAMPLE LDA $35
 1006        STA $37
 1010        RTS
 1005 SAMPLE LDA $35
 1006        STA $37

:RENUMBER
:LIST
 1000 *   LITTLE COPY EXAMPLE
 1010 SAMPLE LDA $35
 1020        STA $37
 1030        RTS
 1040 SAMPLE LDA $35
 1050        STA $37

:COPY 1020,1040,1010
:RENUMBER
:LIST
 1000 *   LITTLE COPY EXAMPLE
 1010        STA $37
 1020        RTS
 1030 SAMPLE LDA $35
 1040 SAMPLE LDA $35
 1050        STA $37
 1060        RTS
 1070 SAMPLE LDA $35
 1080        STA $37
```

Listing Control Commands

```
FAST, SLOW, PRT, *
```

The listing control commands are used to control the speed of display on the screen, and to control printing of listings on other devices. One special command allows sending setup control characters to your printer.

FAST and SLOW Commands:

```
:FAST
:SLOW
```

FAST sets the listing speed to the normal speed, which is too fast for most people to read. When you first enter S-C Macro Assembler, it is already in the FAST mode. If you abort a listing by hitting the RETURN key, the system returns to the FAST mode.

SLOW sets the listing speed slow enough that you can read it as it goes by on your screen.

In both the FAST and SLOW modes, you can momentarily stop the listing by tapping the space bar (or any other key except RETURN). You can abort the listing by typing the RETURN key. When the listing is stopped, pressing two keys at the same time will cause one additional line to be listed.

PRT Command:

```
:PRT
```

Provides a "hook" for a user-supplied printer driver. If you have an Apple parallel or serial printer board, the usual PR#slot will activate your printer. If you have a printer driven through the game port, or an interface board which requires special handling, you can use the PRT command to turn it on. If you don't need it for a printer, PRT can serve as a second USR command.

PRT executes a JSR $1009, where you can put a JMP to your printer driver. A sample printer driver is included on the disk as a source program, called SAMPLE PRINTER DRIVER. Appendix C includes a listing and description of this program. You can examine it to learn how to write your own.

* Command:

```
:*string
```

Sends the specified string to the currently selected output device. If your printer is currently selected, you can send control-codes to it.

Remember that in order to enter a control-character on an input line, you type the control-O (override) followed by the desired character.

Object Commands

ASM, MGO, VAL, SYMBOLS

Object commands are used to assemble source programs into object programs, execute object programs, and to print the value of label expressions after assembly.

ASM Command:

:ASM

Initiates assembly of your source program. The S-C Macro Assembler is a two-pass assembler. During the first pass it builds a symbol table with the definition of every label used in your program. During the second pass the assembler stores object code into memory (or writes it on a disk file) and produces an assembly listing on the screen and/or the printer. At the end of the second pass all the labels and their values are listed in alphabetical order.

The assembly listing may be momentarily interrupted and restarted by tapping the space bar. You may abort the assembly by typing the RETURN key. The assembly listing may also be controlled with the .LIST directives, to print any part of it or none at all.

If any errors are detected in either pass, they are printed along with a copy of the offending line. Assembly normally continues after an error, so that you can catch as many errors as possible in one pass. If any errors are detected during pass one, pass two is not attempted. At the end of assembly the total number of errors is printed. All the assembly error messages with their probable causes are listed in Appendix B.

MGO Command:

:MGO expression

Begins execution of your object program. An expression or label name must follow the MGO command to define the place to begin execution. Remember that an object program is the result of an assembly, so you must have used the ASM command before the MGO will execute your program.

```
:MGO BEGIN        (Start execution at label BEGIN)
:MGO $803         (Start execution at $803)
```

Your program can return to S-C Macro Assembler either by using an "RTS" instruction, by a "JMP $3D0" (if DOS is active), or by a "JMP $1003" ($D003 for language card version). You may also abort your program by hitting the RESET key. If your Apple has the Autostart ROM, you will come out in the assembler. If you come out in the monitor, type 3D0G to reenter the assembler.

In the tape version of the S-C Macro Assembler, the MGO command is named "RUN". The disk version uses "MGO" because the word "RUN" is a DOS command. If you type a RUN command, DOS will attempt to load an Integer BASIC or Applesoft program (possibly clobbering the assembler or the source program in memory).

VAL Command:

:VAL expression

Evaluates any legal operand expression, and prints the value in hexadecimal. It may be used to quickly convert decimal numbers to hexadecimal, to determine the ASCII code for a character, or to find the value of a label from the last assembled program.

```
:VAL 12345
3039
:VAL -21846
AAAA
:VAL 'X
0058
:VAL LOOPA+3
084E
```

SYMBOLS Command:

:SYMBOLS

Displays a copy of the Symbol Table, just like the one that normally is printed at the end of pass two of an assembly.

```
:SYMBOLS
21C2- LIST.SOURCE.IF.LISTING
1CC6- LOAD
1896- LIST.ASM.LINE
.01=18A4, .02=18AC, .04=18B7, .15=18B9
.03=18BF
31A0- MACLBL
```

Miscellaneous Commands

AUTO, MANUAL, INCREMENT, MEMORY, RST, MNTR. USR

The last seven commands do not fit in any other category.

AUTO Command:

```
:AUTO
:AUTO #
```

Turns on automatic line numbering mode. In this mode, a new line number is automatically generated every time you end a line. Lines are ended by typing RETURN, by backspacing over the prompt symbol, and by typing control-X.

If AUTO is used with no parameter, the generated line numbers will start with the next number after the last line you entered or deleted. The next number is formed by adding the current INCREMENT value. The increment can be changed with the INCREMENT command.

AUTO followed by a line number will start the numbering at that parameter.

AUTO should be used when EXEC-ing in text files from another source. This way, you can even use the S-C Macro Assembler to edit BASIC programs which have been listed into text files (as long as you don't need to renumber the BASIC line numbers).

You can type commands while in the AUTO mode by typing backspaces to the beginning of the line (next to, not over, the prompt) and then typing any command. You can leave the AUTO mode by typing the MANUAL command.

The AUTO mode is also terminated by hitting RESET, and after any error message.

MANUAL Command:

```
:MANUAL
```

Terminates the automatic line numbering (AUTO) mode.

INCREMENT Command:

```
:INCREMENT number
```

Sets the increment used for automatic line number generation (both control-I generated numbers and AUTO mode numbers.) The increment is normally 10. but you may set it to any value between 0 and 9999. (Of course, an increment of 0 makes no sense. Neither does a large value like 9999. But you can use them if you wish!)

```
:INC 5        (set increment to 5)
:INC 10       (set increment to 10)
```


MEMORY Command:

```
:MEMORY
```

Displays the beginning and ending memory addresses of the source program and of the symbol table.

```
:MEM
SOURCE PROGRAM: $94F3-9600
  SYMBOL TABLE: $3200-3274
```

Memory between the top of the symbol table and the bottom of the source program is free to be used without clobbering anything.

The assembler automatically protects memory (during assembly) from $1000 to the top of the symbol table, and from the bottom of the source program through $FFFF. This insures that your object program will not clobber the assembler. the source program, or DOS.

MNTR command:

```
:MNTR
```

Enters the Apple system Monitor. This is the same as CALL -151 from BASIC. You may reenter the S-C Macro Assembler by typing 1003G or 3D0G.

```
:MNTR
*
```

RST command:

```
:RST expression
```

Changes the RESET vector to the specified value. If you are using the Autostart Monitor, pressing the RESET key causes a branch to the address in the RESET vector. Normally this is set to $3D0 by DOS to reenter the assembler, but you may change it to enter the monitor, BASIC, or your own program.

```
:RST -151      (RESET enters the monitor)
:RST $FF69     (also enters the monitor)
:RST $3D0      (RESET enters DOS and assembler)
:RST $800      (RESET enters program at $800)
```

USR Command:

```
:USR whatever
```

An open-ended command, waiting for you to design and activate.

When you type the command "USR", a JSR $1006 instruction is executed. If you have not installed a JMP to your own program at $1006, the command is equivalent to a "No Operation" command. You can write a program to process your own command, and put a JMP instruction to it at $1006.

The entire command line is stored in the monitor input buffer, starting at $0200. Your USR command processor can scan the input buffer to pick up any parameters you wish.

Sample USR command processors are published from time to time in the Apple Assembly Line newsletter.

DOS COMMANDS

All the Apple DOS commands are valid, even though you are operating from within S-C Macro Assembler. This feature allows you to maintain your source and object programs on disk using the LOAD, SAVE, BLOAD, and BSAVE commands. Source programs will appear in the disk catalog with a type code of "I", just as though they were Integer BASIC programs.

Housekeeping Commands: CATALOG, RENAME, DELETE, LOCK, UNLOCK, VERIFY, MON, NOMON, and MAXFILES can be used as you desire. They will function exactly the same within S-C Macro Assembler as they do within BASIC.

Source Maintenance Commands: LOAD and SAVE when used with a filename will be interpreted by DOS. If no file name is included, S-C Macro Assembler will interpret them as cassette tape commands.

Object Maintenance Commands: BSAVE, BLOAD, and BRUN commands may be used to maintain object programs on the disk and to execute them. Be careful when using BLOAD and BRUN that the program you are loading does not load on top of anything you want to keep!

I/O Selection Commands: PR#, IN#, and EXEC commands may be used. PR#slot will turn on Apple intelligent interfaces for printers and other output devices. IN#slot may be used with other terminals, modems, et cetera. EXEC will execute a stream of commands or read in a series of source lines from a text file.

BASIC Commands: INT and FP may be used to exit the S-C Macro Assembler and enter either Integer BASIC or Applesoft.

Commands you should not use: RUN, CHAIN, and INIT will not do what you expect. Avoid typing the "RUN filename" command, because it will be recognized by DOS as an attempt to load and execute an Integer BASIC or Applesoft program. However, since the DOS links have been set up for S-C Macro Assembler, the program will not execute. It will just clobber memory, possibly your source program or the assembler itself! The CHAIN command is equally dangerous. INIT will properly format a disk, but it will write your source program (which is not executable) as the HELLO program! It is much better to INIT from within Applesoft or Integer BASIC.

MONITOR COMMANDS

All of the Apple II Monitor commands are available from within S-C Macro Assembler. You use them by typing a dollar sign ($) after the prompt symbol, followed by any monitor command.

Monitor commands are explained on pages 40-66 of the Apple II Reference Manual. With these commands you may examine, change, move or verify memory; read and write cassette tapes; dis-assemble machine language programs; execute programs; and perform hexadecimal arithmetic. If you have the old Monitor ROM, (rather than the Autostart ROM), you may use the trace and single-step debugging features. If you have Integer BASIC in ROM or language card, and it is currently selected, you may call the mini-assembler at $F666.

The availability of all these commands makes it much easier for you to develop and debug assembly language programs.

Chapter 5 -- DIRECTIVES

Twenty assembler directives are available in the S-C Macro Assembler to control the assembly process and to define data in your programs. These are all indicated by a period followed by two or more letters.

.OR	ORigin	.BS	Block Storage
.TA	Target Address	.LIST	Control Assembly LISTing
.TF	Target File	.TI	TItle
.IN	INclude file	.US	USer defined directive
.EN	END of program	.PG	PaGe eject
.EQ	EQuate	.DO	Conditional Assembly
.DA	DAta	.ELSE	Conditional Assembly
.HS	Hex String	.FIN	Conditional Assembly
.AS	Ascii String	.MA	MAcro definition
.AT	Ascii Terminated	.EM	End Macro

Origin:

`.OR expression`

Sets the program origin and the target address to the value of the expression. Program origin is the address at which the object program will be executed. Target address is the memory address at which the object program will be stored during the assembly. The .OR directive sets both of these to the same value, which is the normal way of operating.

If you do not use the .OR directive, the assembler will set both the program origin and the target address to $0800. If the expression is not defined during pass one prior to its use in the .OR directive, an error message is printed.

If a .TF (Target File) was active before the .OR directive, it will be closed out.

Target Address:

`.TA expression`

Sets the target address at which the object code will be stored during assembly. The target address is distinct from the program origin (which is either set by the .OR directive, or is implicitly set to $0800). The .OR directive sets both the origin and the target address; the .TA directive sets only the target address. Object code is produced ready to run at the program origin, but is stored starting at the target address.

When you wish to assemble a program which will execute at an address normally occupied by the assembler ($1000-31FF), the symbol table ($3200 up), or the source program text (bottom of DOS down), you need to use the .TA and .OR directives. Set the origin first, using the .OR directive; then set the target address to a safe value using the .TA directive. It is usually safe to start the target area at $0800, provided your object

code does not exte[illegible] beyond $0PFF. If you are us[illegible] macros, that will take so[illegible] space from $0PFF down. See t[illegible] chapter on macros for details.

```
                    1000 *--------------------------------
                    1010 *    SAMPLE PROGRAM TO ILLUSTRATE
                    1020 *      THE ".TA" DIRECTIVE
                    1030 *--------------------------------
                    1040         .OR $1000
                    1050         .TA $0800
1000- AD 0C 10      1060 DEMO    LDA AVALUE
1003- AE 0D 10      1070         LDX XVALUE
1006- AC 0E 10      1080         LDY YVALUE
1009- 4C 00 10      1090         JMP DEMO
                    1100 *--------------------------------
100C- 0C            1110 AVALUE .DA #12
100D- 22            1120 XVALUE .DA #34
100E- 38            1130 YVALUE .DA #56

SYMBOL TABLE

100C- AVALUE
1000- DEMO
100D- XVALUE
100E- YVALUE

0000 ERRORS IN ASSEMBLY
```

As you can see in the example, the assembly listing looks as though the program were stored at $1000. However, the object code is actually stored at $0800, which is the target address set in the .TA directive. If we dis-assemble memory starting at $0800, we see:

```
:$800L
0800-  AD 0C 10   LDA  $100C
0803-  AE 0D 10   LDX  $100D
0806-  AC 0E 10   LDY  $100E
0809-  4C 00 10   JMP  $1000
```

After the assembly is complete, there are several ways to position the code in memory where it really should be.

1. You can save the object code on cassette using the Apple Monitor "W" command.

2. You can save the object code on disk using the DOS "BSAVE" command. Be sure you do not try to reload it while you are executing the assembler, or you may clobber it!

3. You can use the Monitor's memory move command (addr1<addr2.addr3M). This command will move the block of memory from addr2 through addr3 to the area beginning at addr1. Be sure not to move the code over the top of the assembler unless you first exit from the assembler!

If you ne[illegible] a larger safe area than that given between $0800 and $0PFF, [illegible]ou can patch the assembler at loc[illegible]ion $101D. This location currently contains $32, which is the [illegible]ge number of the start of the assembler's symbol table. If you change this value to $45, for example, the symbol table will start at $4500 instead of $3200. This will leave the area from $3200 through $44FF free for a target area. To be effective, this change should be made before using the ASM command. Be sure to leave enough room between the start of the symbol table and the bottom of DOS for all of your source program as well as the symbol table.

The safest way to handle programs too large to fit comfortably in memory is to use the .TF directive, explained below.

Target File: `.TF filename`

Causes the object code generated to be stored on a binary file, rather than in memory. Only the code which follows the .TF directive will be stored on the file. Code will be stored on the file until another .TF directive is encountered, or until a .TA or .OR directive is encountered.

The filename specifier may include volume, drive, and slot numbers if necessary. If you have both .IN and .TF directives in the same assembly, and the files involved are not on the same disk, you will need to specify drive number (and maybe slot number) with every .IN and with every .TF directive.

If your program consists of several pieces with different origins, and you want them all to be put on files, each piece will require a separate .TF directive. The object code is written on a binary file, which may only have one origin.

During assembly, S-C Macro Assembler temporarily patches DOS to allow a binary file to be handled with text file commands. It also creates a text file with your specified name and uses text file techniques to write the object code into the file. When assembly is complete, or when the .TF range is ended by encountering another .TF (or .TA or .OR), the text file is transformed into a binary file by modifying the DOS directory entry.

If you have typed MON C (a DOS command) before assembly, the DOS commands issued by the assembler for the .TF directive will print on the assembly listing. For each .TF directive, during pass two, you will see the following sequence:

```
OPEN file name
DELETE file name
OPEN file name
WRITE file name
```

If you have typed MON O (a DOS command to render text file output visible), you will see lots of crazy characters on the screen during pass two of the assembly. These are the object code bytes which are being written on the Target File. It is better to not set MON O mode.

Include: `.IN filename`

Causes the contents of the specified source file to be included in the assembly.

The program which is in memory at the time the ASM command is typed is called the "root" program. Only the root program may have .IN directives in it. If you attempt to put .IN directives in an included program, the "NESTED .IN" error will print.

When the .IN directive is processed, the root program is temporarily "hidden" and the included program is loaded. Assembly then continues through the included program. When the end of the included program is reached, it is deleted from memory and the root program is restored. Assembly then continues with the next line of the root program.

If you type the MON C command (a DOS command) before beginning assembly, the LOAD commands issued by the assembler will be printed with the listing. Each included program is loaded in turn during pass one of the assembly, and again during pass two.

The .IN directive is useful in assembling extremely large programs, which cannot fit in memory all at once. It is also useful for connecting together a library of subroutines with a main program. (Some programmers prefer this method over the use of macros.)

The filename portion of the directive is in standard DOS format, and may include volume, slot, and drive number.

End of Program: `.EN`

Defines the end of the source program, or of an included (.IN) module. You would normally make this the last line, but you may place it earlier in order to assemble only a portion of your source program. If no .EN is present anywhere in your program, the assembler will assume you meant to put one after the last line. (This is different from most assemblers, which for some strange reason go completely crazy if the .EN directive is missing!)

Equate: `label .EQ expression`

Defines the label to have the value of the expression. If the expression is not defined, an error message is printed. If you neglect to use a label with an equate directive, an error message is printed also. One common use for this directive is to define all the page-zero variables your program uses.


```
0045-           1000 ACC    .EQ $45
0200-           1010 IN     .EQ $200
0050-           1020 ACL    .EQ $50
0051-           1030 ACH    .EQ ACL+1
C064-           1040 PDL0   .EQ $C064     PADDLE 0


SYMBOL TABLE

0045- ACC
0051- ACH
0050- ACL
0200- IN
C064- PDL0

0000 ERRORS IN ASSEMBLY
```

Data: `label .DA exprlist`

Creates constants or variables in your program. "Exprlist" is a list of one or more expressions separated by commas. Each expression may be treated as one or two bytes, depending on the initial character:

two bytes (low byte first):	expression alone
one byte (low order byte):	#expression
one byte (high order byte):	/expression

The value of the expression, as one or two bytes, is stored at the current location. If a label is present, it is defined as the address where the first byte of data is stored. (If you use .DA to define a variable, it is a good habit to use an expression like "*-*", which has a value of zero. This weird expression will make your program more self-explanatory when you look at it again next year.)

```
0200-           1000 IN     .EQ $200      INPUT BUFFER
0800- E8 03     1010 TEN3   .DA 1000
0802- 64 C0     1020 PDL0   .DA $C064     PADDLE 0 ADDRESS
0804- C1        1030 LTRA   .DA #$C1      ASCII LETTER A
0805- 02        1040 BPPG   .DA /IN       BUFFER PAGE


SYMBOL TABLE

0805- BPPG
0200- IN
0804- LTRA
0802- PDL0
0800- TEN3
```

Hex String: `label .HS hhh...h`

Converts a string of hex digits (hhh...h) to binary, two digits per byte, and stores them starting at the current location. If a label is present, it is defined as the address where the first byte is stored. If you do not have an even number of hexadecimal digits, the assembler prints an error message.

NOTE: Unlike hexadecimal numbers used in operand expressions, you must not use a dollar sign with the .HS directive. Do not let this confuse you.

```
0800- F1          1000          .HS F1
0801- 23 AB 45 1010 STR         .HS 23AB45
0804- 01 23 45
0807- 67 89 AB
080A- CD EF       1020 QT       .HS 0123456789ABCDEF


SYMBOL TABLE

0804- QT
0801- STR

0000 ERRORS IN ASSEMBLY
```

Ascii String: `label .AS daaa...ad`

Stores the binary form of the ASCII characters "aaa...a" in sequential locations beginning at the current location. If a label is present, it is defined as the address where the first character is stored. The string "aaa...a" may contain any number of the printing ASCII characters. You indicate the beginning and end of the string by any delimiter "d" that you choose.

ASCII character codes are seven bit values. The .AS directive normally sets the high-order, or 8th, bit to zero. Some people like to use ASCII codes with the high-order bit set to one, so S-C Assemblers include an option for this.

```
.AS daaa...ad      sets the high-order bits = 0
.AS -daaa...ad     sets the high-order bits = 1
```

This syntax restricts the choice of the delimiter slightly: it may be any printing character other space or minus.

Ascii Terminated: `label .AT daaa...ad`

This works just like the .AS directive, except that the high-order bit of the last byte in the string is set opposite from the preceding bytes. This allows a message-printing routine to easily find the end of a message.

```
                1000 *-------------------- ----------
                1010 *       WITH .AS
                1020 *--------------------------------
0800- 53 54 52
0803- 49 4E 47  1030 STR    .AS "STRING" DELIMITER IS "
0806- 22 22 22  1040 QT     .AS /"""/    DELIMITER IS /
0809- C8 D5 C8
080C- BF        1050 HUH    .AS -QHUH?Q  DELIMITER IS Q
                1060 *--------------------------------
                1070 *       WITH .AT
                1080 *--------------------------------
080D- 53 54 52
0810- 49 4E C7  1090 STR1   .AT "STRING"
0813- 22 22 A2  1100 QT1    .AT /"""/
0816- C8 D5 C8
0819- 3F        1110 HUH1   .AT -QHUH?Q


SYMBOL TABLE

0809- HUH
0816- HUH1
0806- QT
0813- QT1
0800- STR
080D- STR1

0000 ERRORS IN ASSEMBLY
```

Block Storage: `label .BS expression`

Reserves a block of bytes starting at the current location in the program. The expression specifies the number of bytes to reserve. If there is a label, it assigned the value at the beginning of the block.

The address of the beginning of the block will be printed in the address column of the assembly listing.

If the object code is being stored directly into memory, no bytes are stored for the .BS directive. However, if the object code is beirg written on a file using the .TF directive, the .BS directive will write <expression> bytes on the file. All the bytes written will have the value $00.

```
0800- 34 12     1000 A      .DA $1234
0802-           1010 B      .BS 2
                1020 *--------------------------------
0804- AD 00 08  1030        LDA A
0807- 8D 02 08  1040        STA B
080A- AD 01 08  1050        LDA A+1
080D- 8D 03 08  1060        STA B+1


SYMBOL TABLE

0800- A
0802- B

0000 ERRORS IN ASSEMBLY
```

Title: `.TI expression,title`

When .TI is in effect the assembly listing will have a title line and page number at the top of each page. The expression specifies the maximum number of lines you want to print on each page. The title can be up to 70 characters long and will be printed starting at the left margin. " PAGE xxxx" will be printed immediately after the title. If there is no title the page number will be printed at the left margin. Spacing or centering of the title and page number can be adjusted by adding leading or trailing spaces to the title.

The S-C Macro Assembler will issue an automatic formfeed when a page fills up. If you want to end a page early, use the .PG directive. You can use more than one .TI directive in a program if you like. The .TI directive also issues a formfeed command.

You can turn off titling by using .TI with a pagelength of zero.

Listing Control: `.LIST optionlist`

Controls the listing output of the assembler. "Optionlist" is a list of one or more of the following keywords:

OFF	Listing off.
ON	Listing on.
MOFF	Macro expansion listing off.
MON	Macro expansion listing on.

If .LIST OFF is put at the beginning of the source program, and no .LIST ON is used, no listing at all will be produced. The program will assemble much faster without a listing, as most of the time is consumed in putting characters on the screen and scrolling the screen up.

If you put .LIST OFF at the beginning of your source program, and .LIST ON at the end, only the alphabetized symbol table will print.

You may also use this pair of directives to bracket any portion of the listing you wish to see or not see.

With .LIST MON in effect, the complete macro expansion will be listed. The call line will be printed with its line number, then the expansion lines, each with a line number of "0000>".

Page Control: `.PG`

Prints an ASCII Form Feed character ($0C). If the assembly listing is being printed on a printer which recognizes this character, a form feed will occur and the next listing line will appear at the top of the next page. The .PG line itself is not listed.

Conditional Assembly:

```
.DO expression
.ELSE
.FIN
```

With these directives, you can include or exclude a particular section of code in the assembly, depending on a condition set earlier. The operand expression is evaluated as a truth value, and must be defined before the .DO. Zero means skip lines, non-zero means assemble them.

The .ELSE directive toggles the current truth value, allowing an if...then...else kind of structure. There may be more than one .ELSE directive within the .DO - .FIN block; each time .ELSE is encountered the truth value is switched. .FIN terminates the conditional section. .ELSE is optional but .FIN is required.

.DO - .FIN blocks may be nested, up to 8 deep.

These directives are often used to produce different specialized versions of a program from the same source code. For example, the main memory and language card versions of S-C MACRO Assembler were assembled from the same source file, using a .DO flag called LCASM. When a change is made to the assembler, we only have to edit one source line to generate the two different versions. We assemble it twice: once with LCASM=1, once with LCASM=0.

.DO - .FIN blocks can also be used to exclude testing routines from the finished program, and to add or delete extra variables.

```
:LIST

 1000 *   CONDITIONAL ASSEMBLY DEMO
 1010 *--------------------------------
 1020 FLAG    .EQ 0
 1030         .DO FLAG
 1040         JSR SOMEPLACE
 1050         .ELSE
 1060         JSR ANOTHER.PLACE
 1070         .ELSE
 1080         JSR ONE.MORE.PLACE
 1090         .FIN
 1100         RTS
 1110 *--------------------------------
 1120 SOMEPLACE      RTS
 1130 ANOTHER.PLACE  RTS
 1140 ONE.MORE.PLACE RTS

:ASM

                1000 *   CONDITIONAL ASSEMBLY DEMO
                1010 *--------------------------------
0000-           1020 FLAG    .EQ 0
                1030         .DO FLAG
                1050         .ELSE
0800- 20 05 08  1060         JSR ANOTHER.PLACE
                1070         .ELSE
                1090         .FIN
0803- 60        1100         RTS
                1110 *--------------------------------
0804- 60        1120 SOMEPLACE      RTS
0805- 60        1130 ANOTHER.PLACE  RTS
0806- 60        1140 ONE.MORE.PLACE RTS

:1020 FLAG    .EQ 1

:ASM

                1000 *   CONDITIONAL ASSEMBLY DEMO
                1010 *--------------------------------
0001-           1020 FLAG    .EQ 1
                1030         .DO FLAG
0800- 20 07 08  1040         JSR SOMEPLACE
                1050         .ELSE
                1070         .ELSE
0803- 20 09 08  1080         JSR ONE.MORE.PLACE
                1090         .FIN
0806- 60        1100         RTS
                1110 *--------------------------------
0807- 60        1120 SOMEPLACE      RTS
0808- 60        1130 ANOTHER.PLACE  RTS
0809- 60        1140 ONE.MORE.PLACE RTS
```

Macro Definition: `.MA acro name`
End Macro: `.EM`

A macro definition must begin with the directive .MA <macro name> and end with the .EM directive. For detailed informatio see the chapter on macros.

User Directive: `label .US whatever`

To allow for possible expansion of the assembler by users, the .US directive has been included. When the opcode is processed it will branch to $100C. That location normally contains a JM instruction, which treats the .US as a comment. The source line will be in the system buffer starting at $0200 (without the line number).

If you desire to use the .US directive, change $100C-$100E to jump to your own program. Some details of the steps necessary to implement your own directives are published in the September, 1981 issue of Apple Assembly Line, pages 12-15. Yo may also disassemble S-C Macro Assembler, if you wish, and examine the existing directives.

Chapter 6 -- OPERAND EXPRESSIONS

Operand expressions are written using elements and operators. The valid operators are +, -, *, /, <, =, and >. Terms may be decimal or hexadecimal numbers, labels, a literal ASCII character, or an asterisk (*). The first term in an expression may be preceded by a + or - sign.

ELEMENTS

Decimal Numbers: Any number in the range from 0 through 65535, written in the normal way.

```
0800- A9 C8      1000         LDA #200
0802- A2 F6      1010         LDX #-10
0804- 6B 8B      1020         .DA 35691
FFFF-            1030 FLAG    .EQ -1
```

Hexadecimal Numbers: Any number in the range from $0 through $FFFF. Hexadecimal numbers are indicated by a preceding dollar sign, and may have from one to four digits.

```
                 1050         .OR $880
0880-  A9 2F     1060         LDA #$2F
0882-  85 CA     1070         STA $CA
0884-  D0 1D     1080         BNE $8A3
0886-  20 2A E0  1090         JSR $E02A
00AB-            1100 VALL    .EQ $AB
1278-            1110 NUM     .EQ $1278
0888-  78 12     1120 DATA    .DA $1278
```

Beware of leaving out the dollar sign; the assembler may be quite satisfied to think of your hexadecimal number as a decimal one if you omit the $. In some cases even a number with letters in it, such as 23AB, may be acceptable; it may be interpreted as decimal 23 and a comment "AB".

Labels: There are three types of labels in S-C Macro Assembler. Normal labels are from 1 to 32 characters long. The first character must be a letter; following characters may be letters, digits, or periods. Local labels are written as a period followed by one or two digits. Private labels are written as a colon followed by one or two digits.

Labels must be defined somewhere if they are to be used in an expression. Labels used in operand expressions after .OR, .TA, .BS, and .EQ directives must be defined prior to use (to prevent an undefined or ambiguous location counter). Labels are defined by being written in the label field of an instruction or directive line.

In previous versions of S-C Assemblers, special care was necessary to assure that all zero-page labels were defined prior to their use in the address fields of certain 6502 instructions. This care is no longer needed, because the S-C Macro Assembler handles forward references properly in such cases.

Literal ASCII Characters: Literal characters are written as an apostrophe followed by the character. The value is the ASCII code of the character (a value from $00 through $7F).

```
0041-           1000 LTRA   .EQ 'A
0930- 58 41 00  1010        .DA #'X,'A
0933- C9 5A     1020        CMP #'Z
```

If you wish to use literal ASCII values with the sign bit equal to 1 (codes $80-$FF), you can do so by adding $80 in the operand expression:

```
00C1-           1000 LTRA   .EQ 'A+$80
0930- D8 C1 00  1010        .DA #'X+$80,'A+$80
0933- C9 DA     1020        CMP #'Z+$80
```

Asterisk (*): Stands for the current value of the location counter. This is useful for storing the length of a string as a constant in a program. I also use it in filling up to the end of the page to assure that following code begins at an even page boundary.

```
080A- 08        1070 QT     .DA #QTSZ
080B- 41 4E 59
080E- 20 4D 45
0811- 53 53 41
0814- 47 45     1080        .AS /ANY MESSAGE/
0008-           1090 QTSZ   .EQ *-QT-1
0816- 00 00     1100 VAR    .DA *-*
0818-           1110 FILLER .BS $900-*   FILL UP
                1120 *                   THROUGH $8FF
```


OPERATORS

You can use arithmetic and relational operators in operand expressions. Expressions are evaluated strictly from left to right, with no other precedence implied. Parentheses cannot be used to change this order.

Arithmetic Operators (+-*/): Any of the four arithmetic operators may be used in an operand expression.

All operations are performed on 16-bit values. Multiplication returns the low-order 16-bits of the 32-bit product.

Overflow and division-by-zero are not considered assembly errors. Overflow merely truncates, returning the low-order 16-bits. Division-by-zero returns the value $FFFF (65535).

Relational Operators (<=>): The three relational operators compare two 16-bit values. If the relation is true, the result is 1. If the relation is false, the result is 0. The result can be used in further calculations, and as the truth value for conditional assembly (.DO directive).

Only the three elementary operators are available: less than (<), equal (=), and greater than (>). They cannot be combined as they are in BASIC to form <=, <>, or >=.

The result of a relational expression is a true or false value. A value of zero is considered to be false, and a non-zero value is considered to be true. You may operate on logical values with * and + operators: * has the effect of the logical AND, and + has the effect of the logical OR operation.

If you are in doubt how an expression will evaluate, you can use the VAL command to see. Or you can go ahead and assemble your program and see how it turns out.

Chapter 7 -- MACROS

A macro is a single instruction in your source code, which when assembled is replaced by a defined series of instructions. With macros, you can use a shorthand for commonly used sequences, effectively define your own instructions for the 6502, or even rename the 6502 opcodes.

A Simple Macro

How often do you increment a 16-bit variable like this:

```
1100         INC PTR     INCREMENT LOW BYTE
1110         BNE .1      CARRY?
1120         INC PTR+1   YES, INCREMENT HIGH BYTE
1130 .1      (whatever)  NO, GO ON
```

We can define a macro called INCD to do all that. Just put this definition at the beginning of the program:

```
1000         .MA INCD    MACRO NAME
1010         INC ]1      CALL PARAMETER
1020         BNE :1      PRIVATE LABEL
1030         INC ]1+1
1040 :1
1050         .EM         END OF DEFINITION
```

Now you only need to enter:

```
1100         >INCD PTR
1110         (whatever)
```

The object code will be the same in either case. If an operation is used only once or twice in a program, it isn't really worth the effort to define a macro for it; but if you have to do the same operation on several different variables, a macro can save a lot of work. It can also help prevent common mistakes, such as mixing up the high- and low-bytes of 16-bit variables.

Now to explain that definition. The directive .MA signals the start of a macro definition, and is followed by the macro name. The operand "]1" is a macro call parameter. In the assembly, it will be replaced by the operand in the macro call line (in our example, PTR). The label ":1" is a private label used to name a branch point within the macro. The directive .EM signals the end of a macro definition. A macro must be defined before it is called, so it is best to put all macro definitions together at the beginning of the program.

Once you have defined a macro, it can be called at any time by typing >name in the opcode field and any parameters needed in the operand field. At assembly time, the assembler will insert the correct code from the macro definition.

```
                1000 *--------------------------------
                1010 *      DEMONSTRATE CONDITIONAL ASSEMBLY IN
                1020 *--------------------------------  MACRO
                1030         .MA INCD
                1040         .DO ]#=2
                1050         INC ]1,]2
                1060         BNE :3
                1070         INC ]1+1,]2
                1080 :3
                1090         .ELSE
                1100         INC ]1
                1110         BNE :3
                1120         INC ]1+1
                1130 :3
                1140         .FIN
                1150         .EM
                1160 *--------------------------------
0800-           1170         >INCD $12
                0000>         .DO 1=2
                0000>         .ELSE
0800- E6 12     0000>         INC $12
0802- D0 02     0000>         BNE :3
0804- E6 13     0000>         INC $12+1
                0000> :3
                0000>         .FIN
0806-           1180         >INCD $1234
                0000>         .DO 1=2
                0000>         .ELSE
0806- EE 34 12  0000>         INC $1234
0809- D0 03     0000>         BNE :3
080B- EE 35 12  0000>         INC $1234+1
                0000> :3
                0000>         .FIN
080E-           1190         >INCD $12,X
                0000>         .DO 2=2
080E- F6 12     0000>         INC $12,X
0810- D0 02     0000>         BNE :3
0812- F6 13     0000>         INC $12+1,X
                0000> :3
                0000>         .ELSE
                0000>         .FIN
0814-           1200         >INCD $1234,X
                0000>         .DO 2=2
0814- FE 34 12  0000>         INC $1234,X
0817- D0 03     0000>         BNE :3
0819- FE 35 12  0000>         INC $1234+1,X
                0000> :3
                0000>         .ELSE
                0000>         .FIN
```


Nested Macro Definitions

You can call macros within macro definitions. I do not recommend it, but many programmers delight in the intricacies of nested and recursive macros.

Suppose you want to write a macro which can be used to call one or more subroutines on a single source line. For example, CALL SAM should translate to JSR SAM. CALL SAM,TOM should generate two JSR's, and so on. You could do it at least two ways: using conditional directives, or using nested macro definitions.

Using conditional directives is fairly straightforward. The following program shows how. The]# parameter is tested to determine whether another parameter is present, and if so a JSR line is produced.

```
                1000         .MA CALL
                1010         JSR ]1
                1020         .DO ]#>1
                1030         JSR ]2
                1040         .FIN
                1050         .DO ]#>2
                1060         JSR ]3
                1070         .FIN
                1080         .EM
0800-           1090         >CALL SAM,TOM,JOE
0800- 20 0F 08  0000>         JSR SAM
                0000>         .DO 3>1
0803- 20 11 08  0000>         JSR TOM
                0000>         .FIN
                0000>         .DO 3>2
0806- 20 10 08  0000>         JSR JOE
                0000>         .FIN
0809-           1100         >CALL SAM,TOM
0809- 20 0F 08  0000>         JSR SAM
                0000>         .DO 2>1
080C- 20 11 08  0000>         JSR TOM
                0000>         .FIN
                0000>         .DO 2>2
                0000>         .FIN
080F- 60        2000 SAM     RTS
0810- 60        2010 JOE     RTS
0811- 60        2020 TOM     RTS
```

The other approach uses a nested macro definition. I have set up three separate macros, one for each possible number of parameters: CALL1 for one parameter, CALL2 for two, and CALL3 for three. Then I defined CALL to call the appropriate one of those.

```
                1000         .MA CALL1
                1010         JSR ]1
                1020         .EM
                1030 *--------------------------------
                1040         .MA CALL2
                1050         JSR ]1
                1060         JSR ]2
                1070         .EM
                1080 *--------------------------------
                1090         .MA CALL3
                1100         JSR ]1
                1110         JSR ]2
                1120         JSR ]3
                1130         .EM
                1140 *--------------------------------
                1150         .MA CALL
                1160         >CALL]# ]1,]2,]3
                1170         .EM
                1180 *--------------------------------
0800-           1190         >CALL SAM
0800-           0000>         >CALL1 SAM,,
0800- 20 12 08  0000>>         JSR SAM
0803-           1200         >CALL SAM,JOE
0803-           0000>         >CALL2 SAM,JOE,
0803- 20 12 08  0000>>         JSR SAM
0806- 20 13 08  0000>>         JSR JOE
0809-           1210         >CALL SAM,JOE,TOM
0809-           0000>         >CALL3 SAM,JOE,TOM
0809- 20 12 08  0000>>         JSR SAM
080C- 20 13 08  0000>>         JSR JOE
080F- 20 14 08  0000>>         JSR TOM
                1215 *--------------------------------
0812- 60        1220 SAM     RTS
0813- 60        1230 JOE     RTS
0814- 60        1240 TOM     RTS

SYMBOL TABLE

0813- JOE
0812- SAM
0814- TOM
```


Possible Errors

What happens if you have more parameters on a macro call line than the macro definition expects? The extra parameters are simply ignored. You can use the]# parameter with conditional assembly directives (.DO, .ELSE, and .FIN) to test for the correct number if you wish.

And what if you do not have enough parameters on the call line? The missing ones will be treated as null strings. Again, you may test for the correct number if you wish using conditional assembly directives.

There are three error conditions that the S-C Macro Assembler tests for. If you call a macro that has not been defined earlier in the program, you will see "*** UNDEFINED MACRO ERROR". If you use a .MA directive with no name in the operand field, you will get "*** NO MACRO NAME ERROR". If you use the "]" character in a macro definition without a digit 1-9 or "#" character following, you will get "*** BAD MACRO PARAMETER ERROR".

Sample Macros in MACRO LIBRARY

The file named "MACRO LIBRARY" on the S-C Macro Assembler disk contains more examples of macro definitions. There are at least two ways you can use some of these macros in your programs. The easiest way is to include them with ".IN MACRO LIBRARY" at the beginning of your source program. This technique wastes memory for the unused macros in the source code and in the symbol table, but that's no problem unless your program is very large.

A second way to use them is to first LOAD MACRO LIBRARY, then DELETE the lines containing the definitions you don't need, RENUMBER, and start entering your program.

performed. As each instruction is performed, the PC-register is updated to point at the next instruction in line. Some instructions modify the PC-register directly to change the order of operations (branches and jumps).

The 8-bit registers are called A, X, Y, S, and P. The A-register is an accumulator register. It is the register used in performing addition, subtraction, and logical operations.

The X- and Y-registers are index registers. As such, they are used in computing the effective address of indexed instructions.

The S-register holds the address of the current top-of-stack. The "stack" consists of the memory addresses $0100 through $01FF. Certain operations "push" a byte onto the stack by storing the byte at the location the S-register points to, and then decrementing the S-register. Other operations "pull" a byte off the stack by incrementing the S-register, and then reading the byte that the S-register points to.

The P-register contains seven status bits. (One bit is unused.) Four of these bits can be tested by conditional branch instructions. The entire P-register can be loaded from the stack (PLP instruction) or copied onto the stack (PHP instruction). Some of the bits can be set and cleared directly; some are indirectly affected by the results of operations. The bits are arranged like this:

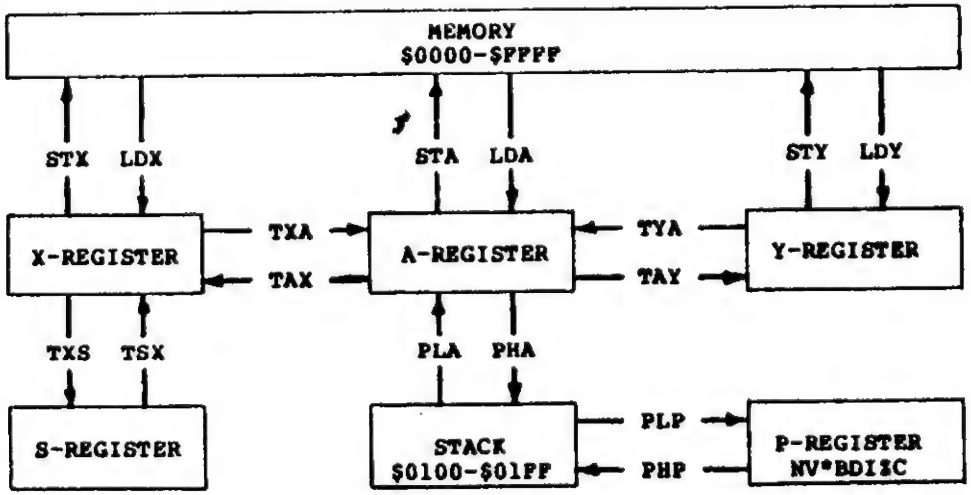
```
7  6  5  4  3  2  1  0

N  V  *  B  D  I  Z  C
```

The bits have the following meaning:

N -- Negative	0 if last result +, 1 if -
V -- Overflow	0 if last result did not overflow, and after CLV instruction; 1 if last result did overflow
* -- Unused	
B -- Break Status	1 after a BRK instruction
D -- Decimal Mode	0 after CLD instruction, 1 after SED
I -- Interrupt	0 after CLI instruction (IRQ interrupt enabled), 1 after SEI instruction (IRQ interrupt disabled)
Z -- Zero	0 if last result non-zero, 1 if last result zero.
C -- Carry	0 if last result did not carry, and after CLC instruction; 1 if last result did carry, and after SEC

The diagram below illustrates the data flow between the registers and memory.



Addressing Modes

One of the features of the 6502 microprocessor which makes it so powerful is its great variety of addressing modes. There are thirteen different modes in all, although no single opcode can use every one of them. The charts later in this chapter show which modes can be used with each opcode. But first, here is a chart showing an example of each mode and the way it is written in assembly language.

Mode	Example	
Implied	DEY	(at least two blanks
Accumulator Mode	ASL	(before comments
Relative Mode	BEQ label	
Immediate Mode	LDA #expr	(low-order byte)
	LDA /expr	(high-order byte)
Direct Modes		
Not Indexed		
Zero Page	LDA expr	(The assembler
Absolute	LDA expr	(uses zero-page
Indexed by X		(form if possible;
Zero Page	LDA expr,X	(if not possible,
Absolute	LDA expr,X	(the assembler
Indexed by Y		(uses the absolute
Zero Page	LDA expr,Y	(form.
Absolute	LDA expr,Y	(
Indirect Modes		
Not Indexed	JMP (expr)	
Pre-Indexed Indirect	LDA (expr,X)	
Post-Indexed Indirect	LDA (expr),Y	

Implied Mode: The address is implied by the nature of the instruction; the operand field is left blank. You need at least two blanks after the opcode if there are comments on the same line. All of the opcodes in this class are only one byte long. They are:

BRK	DEX	PHA	RTS	TAY
CLC	DEY	PHP	SEC	TSX
CLD	INX	PLA	SED	TXA
CLI	INY	PLP	SEI	TXS
CLV	NOP	RTI	TAX	TYA

Relative Mode: Used only by the conditional branch instructions. The expression is converted to a signed offset from the location following the branch instruction. The result must be in the range -128 through +127 to be legal. When the instruction is executed, if the condition tested is true then the one-byte signed offset is added to the contents of the PC-register to get the address of the next instruction to be executed. All of the branch instructions are two bytes long. They are:

BCC	BEQ	BNE	BVC
BCS	BMI	BPL	BVS

Accumulator Mode: Only used by the four shift instructions. These four instructions can also use some of the more complex addressing modes. Each of the shift instructions uses only one byte when in the accumulator mode. The contents of the A-register are shifted, so no memory address is needed. The four shift instructions are:

```
ASL     LSR     ROL     ROR
```

Immediate Mode: Used by eleven of the opcodes. In this mode the operand is the actual value used, rather than the address of a value. For example, "LDA $10" means to load the contents of memory location $0010 into the A-register. On the other hand, "LDA #$10" means to put the value $10 into the A-register. Another way of looking at this is that the "effective address" in immediate mode is the address of the second byte of the instruction. All of the immediate mode opcodes use two bytes. They are:

```
ADC     AND     CMP     CPX     CPY
EOR     LDA     LDX     LDY     ORA
SBC
```

Direct Modes: A one- or two-byte address follows the opcode byte. A two-byte address specifies an address in memory from $0000 to $FFFF. If the address is only one byte long, the memory addressed is assumed to be in page-zero (from $0000 to $00FF).

If the opcode indicates indexing by X, the contents of the X-register are added to the one- or two-byte value from the instruction. The sum is used as the effective address for the operation.

If the opcode indicates indexing by Y, the contents of the Y-register are added to the one- or two-byte value from the instruction. The sum is used as the effective address for the operation.

Indirect Modes: A one- or two-byte address follows the opcode byte. The address is used to pick up two consecutive bytes which in turn are used in the effective address computation.

A two-byte address specifies an address in memory from $0000 to $FFFF. The two-byte form is only usable with one instruction: JMP. It is written as "JMP (expr)". The contents at the address "expr" and "expr+1" are put into the PC-register.

The one-byte address forms specify an address in page zero ($0000 to $00FF). That location and the following one are expected to contain the two-bytes of an address which is used in the effective address computation. There are two modes: pre-indexed indirect and post-indexed indirect.

Pre-indexed indirect, written as "(expr,X)", adds the contents of the X-register to "expr" to get the address of the two bytes which contain the effective address. This mode is hardly ever used. In fact, the only times I have used it are in the special case where the value in the X-register is zero. If (X)=0, then the effective address is the same as if no indexing were performed.

Post-indexed indirect, written as "(expr),Y", looks up the address in the two bytes pointed to by "expr" and "expr+1", and adds the contents of the Y-register to that address. This mode is frequently used to load and store bytes out of a table of bytes. For example, if I have a table of 15 bytes starting at MYTABLE, I can pick up the third byte and store it in the 9th byte like this:

```
1000        LDA #MYTABLE
1010        STA $64         MAKE A POINTER TO MYTABLE
1020        LDA /MYTABLE    IN $64 AND $65
1030        STA $65
1040        LDY #2          POINT AT THIRD BYTE
1050        LDA ($64),Y     GET THAT THIRD BYTE
1060        LDY #8          POINT AT NINTH BYTE
1070        STA ($64),Y     STORE INTO NINTH BYTE
```

The chart on the next page shows which instructions use each of the various direct and indirect modes.

	ACCUM-ULATOR	IMMED-IATE	DIRECT				INDIRECT			
				INDEXED				INDEXED		
	blank	#expr /expr	*expr ZP /ABS	*expr,X ZP /ABS	*expr,Y ZP /ABS	(expr)	(expr,X)	(expr),Y
ADC	--	69	65 / 6D	75 / 7D	-- / 79	--	61	71
AND	--	29	25 / 2D	35 / 3D	-- / 39	--	21	31
ASL	0A	--	06 / 0E	16 / 1E	-- / --	--	--	--
BIT	--	--	24 / 2C	-- / --	-- / --	--	--	--
CMP	--	C9	C5 / CD	D5 / DD	-- / D9	--	C1	D1
CPX	--	E0	E4 / EC	-- / --	-- / --	--	--	--
CPY	--	C0	C4 / CC	-- / --	-- / --	--	--	--
DEC	--	--	C6 / CE	D6 / DE	-- / --	--	--	--
EOR	--	49	45 / 4D	55 / 5D	-- / 59	--	41	51
INC	--	--	E6 / EE	F6 / FE	-- / --	--	--	--
LDA	--	A9	A5 / AD	B5 / BD	-- / B9	--	A1	B1
LDX	--	A2	A6 / AE	-- / --	B6 / BE	--	--	--
LDY	--	A0	A4 / AC	B4 / BC	-- / --	--	--	--
LSR	4A	--	46 / 4E	56 / 5E	-- / --	--	--	--
ORA	--	09	05 / 0D	15 / 1D	-- / 19	--	01	11
ROL	2A	--	26 / 2E	36 / 3E	-- / --	--	--	--
ROR	6A	--	66 / 6E	76 / 7E	-- / --	--	--	--
SBC	--	E9	E5 / ED	F5 / FD	-- / F9	--	E1	F1
STA	--	--	85 / 8D	95 / 9D	-- / 99	--	81	91
STX	--	--	86 / 8E	-- / --	96 / --	--	--	--
STY	--	--	84 / 8C	94 / --	-- / --	--	--	--
JMP	--	--	-- / 4C	-- / --	-- / --	6C	--	--
JSR	--	--	-- / 20	-- / --	-- / --	--	--	--

Instructions

The 56 instructions which the 6502 microprocessor understands can be divided into ten classes:

Transfer Operations
LDA, STA, LDX, STX, LDY, STY
TXA, TAX, TYA, TAY, TXS, TSX
PLA, PHA, PLP, PHP

Arithmetic Operations
ADC, SBC
INC, INX, INY
DEC, DEX, DEY

Logical Operations
AND, ORA, EOR, BIT

Shift Operations
ASL, LSR, ROL, ROR

Compare Operations
CMP, CPX, CPY

Status Operations
SEC, SED, SEI
CLC, CLV, CLD, CLI

Conditional Branch Operations
BCC, BVC, BNE, BPL
BCS, BVS, BEQ, BMI

Unconditional Jump Operations
JMP, JSR

Return Operations
RTS. RTI

Other Operations
NOP, BRK

I will now try to very briefly describe each of the instructions. In the following descriptions, I will use these symbols:

A	A-register
C	carry status bit
D	decimal mode status bit
I	interrupt mask status bit
M	effective address
N	negative status bit
S	S-register (stack pointer)
V	overflow status bit
X	X-register
Y	Y-register
Z	zero status bit
<--	replacement (thing described on right goes into place named on left)
-->	replacement (thing described on left goes into place named on right)
( )	"the contents of" the place named between the parentheses

Transfer Operations: These operators move one byte of data from one register to another, or between registers and memory.

LDA A <-- (M). Load the byte at the effective address into the A-register. Affects N and Z status bits.

LDX X <-- (M). Load the byte at the effective address into the X-register. Affects N and Z status bits.

LDY Y <-- (M). Load the byte at the effective address into the Y-register. Affects N and Z status bits.

STA M <-- (A). Store the A-register at the effective address. Does not change the A-register or status.

STX M <-- (X). Store the X-register at the effective address. Does not change the X-register or status.

STY M <-- (Y). Store the Y-register at the effective address. Does not change the Y-register or status.

TAX (A) --> X. Copies the A-register into the X-register. Affects the N and Z status bits.

TAY (A) --> Y. Copies the A-register into the Y-register. Affects the N and Z status bits.

TXA (X) --> A. Copies the X-register into the A-register. Affects the N and Z status bits.

TYA (Y) --> A. Copies the Y-register into the A-register. Affects the N and Z status bits.

TSX (S) --> X. Copies the stack pointer into the X-register. Affects the N and Z status bits.

TXS (X) --> S. Copies the X-register into the stack pointer. Does not affect status.

PHA M(S) <-- (A), S <-- (S) - 1. Push the A-register onto the stack. Does not affect status.

PHP M(S) <-- (P), S <-- (S) - 1. Push the status register onto the stack. Does not affect status.

PLA S <-- (S) + 1, A <-- M(S). Pull a byte from the stack into the A-register. Affects the N and Z status bits.

PLP S <-- (S) + 1, P <-- M(S). Pull a byte from the stack into the status register.

Arithmetic Operations: These operations perform addition and subtraction on register or memory contents. The operations on the A-register affect the N, Z, and C status bits; the operations on memory and the X- and Y-registers only affect the N and Z status bits.

ADC A <-- (A) + (M) + (C). Adds the byte at the effective address and the carry status to the byte in the A-register. The result is stored in the A-register. If the addition produces a value greater than $FF, carry is set (1 --> C); otherwise carry is cleared (0 --> C).

SBC A <-- (A) - (M) +(C) - 1. Subtracts the byte at the effective address and the "borrow" from the value in the A-register. The result is stored in the A-register. If the subtraction requires a borrow, the carry bit will be cleared (0 --> C); otherwise carry will be set (1 --> C). ("Borrow" is the complement of "carry"; before beginning a subtraction, set carry to 1.)

INC M <-- (M) + 1. Increments (adds one to) the byte at the effective address. Does not change the carry status.

INX X <-- (X) + 1. Increments (adds one to) the value in the X-register. Does not change the carry status.

INY Y <-- (X) + 1. Increments (adds one to) the value in the Y-register. Does not change the carry status.

DEC M <-- (M) - 1. Decrements (subtracts one from) the value at the effective address. Does not change the carry status.

DEX X <-- (X) - 1. Decrements (subtracts one from) the value in the X-register. Does not change the carry status.

DEY Y <-- (X) - 1. Decrements (subtracts one from) the value in the Y-register. Does not change the carry status.

Logical Operations: These operations perform bit-by-bit logical combinations between the operand and the byte in the A-register. All four of the operations affect the N and Z status bits.

AND A <-- (A) and (M). Forms the logical product of the byte at the effective address and the byte in the A-register, leaving the result in the A-register. For each bit position, the following table gives the result value for each combination of operand bits:

	0	1
0	0	0
1	0	1

EOR A <-- (A) eor (M). Forms the exclusive-or result of the byte at the effective address with the byte in the A-register, leaving the result in the A-register. For each bit position. the following table gives the result value for each combination of operand bits:

	0	1
0	0	1
1	1	0

ORA A <-- (A) or (M). Forms the inclusive-or result of the byte at the effective address with the byte in the A-register, leaving the result in the A-register. For each bit position, the following table gives the result value for each combination of operand bits:

	0	1
0	0	1
1	1	1

BIT Z <-- (A) and (M),
N <-- (M) bit 7,
V <-- (M) bit 6. Forms the logical product of the byte at the address and the A-register, setting the Z-flag to the result, but discarding the result. Also sets the N- and V-flags to the values of the corresponding bits of the byte at the address.

Shift Operations: These four operations shift the contents of the A-register or of a memory location by one bit position left or right.

ASL C <-- [7.....0] <-- 0. Arithmetic shift left. Bit 7 of the selected byte is copied into C, bit 6 into bit 7, and so on. A zero is shifted into bit 0.

LSR 0 --> [7.....0] --> C. Logical shift right. Bit 0 of the selected byte is copied into C, bit 1 into bit 0, and so on. A zero is shifted into bit 7.

ROL C <-- [7.....0] <-- C. Rotate left. Circular shift the selected byte and C one bit to the left.

ROR C --> [7.....0] --> C. Rotate right. Circular shift the selected byte and C one bit to the right.

Compare Operations: These operations compare the byte at the effective address with register contents. The comparison is done by subtracting the memory byte from the register byte; the N, Z, and C status bits are affected, but the difference is not stored anywhere.

If the two values compared are equal, the Z status bit will be set. If the two values are considered as unsigned values (0-255), carry will be set if (A) is larger than or equal to (M). If the difference was in the range from $80 to $FF, the N status bit will be set.

CMP N,Z,C <-- (A) - (M). Subtract the byte at the effective address from the byte in the A-register. Set the condition flags, but discard the result.

CPX N,Z,C <-- (X) - (M). Subtract the byte at the effective address from the byte in the X-register. Set the condition flags, but discard the result.

CPY N,Z,C <-- (Y) - (M). Subtract the byte at the effective address from the byte in the Y-register. Set the condition flags, but discard the result.

Status Operations: These operations will directly set or clear specific status bits. The C and V bits are also affected indirectly by other operations.

CLC 0 --> C. Clear carry (or set borrow).

SEC 1 --> C. Set carry (or clear borrow).

CLD 0 --> D. Clear the decimal arithmetic mode. (ADC and SBC will perform normal binary arithmetic.)

SED 1 --> D. Set the decimal arithmetic mode. (ADC and SBC will perform packed BCD arithmetic.

CLI 0 --> I. Clear the IRQ interrupt inhibit status bit. (Allow IRQ interrupts.)

SEI 1 --> I Set the IRQ interrupt inhibit status bit. (Do not allow IRQ interrupts.)

CLV 0 --> V. Clear the overflow status bit.

Conditional Branch Operations: These operations use the relative addressing mode. The branch is taken if the condition tested is true; otherwise execution proceeds with the next instruction in sequence.

BCC Branch if carry clear (C=0). After a compare operation, if the bytes compared are considered to be unsigned values, BCC is equivalent to "branch if less than".

BCS Branch if carry set (C=1). After a compare operation, if the bytes compared are considered to be unsigned values, BCS is equivalent to "branch if greater than or equal".

BEQ Branch if equal (Z=1). After a transfer operation BEQ will branch if the value transfered was zero. After a compare operation. BEQ will branch if the two values compared were equal to each other.

BMI Branch if minus (N=1).

BNE Branch if not equal (Z=0).

BPL Branch if plus (N=0).

BVC Branch if overflow clear (V=0). Overflow is set by an ADC or SBC instruction when the carries out of bit 6 and 7 are not equal. It is cleared when they are equal, or by a CLV instruction. Overflow is also set equal to bit 6 of the byte addressed by a BIT instruction.

BVS Branch if overflow set (V=1).

Uncondit' al Jump Operations: These two operations transfer the effe .ve address to the PC-register unconditionally.

JMP Jump to the effective address. May be direct or indirect.

JSR Jump to a subroutine at the effective address. Before jumping, push the current program counter address (PC) on the stack.

Return Operations:

RTS Return from a subroutine. Pulls the return address from the stack, adds one to it, and branches to that address.

RTI Return from an interrupt. Pulls the status from the stack, and then the return address. Unlike RTS, the return address is not incremented before branching.

Other Operations:

NOP No operation. Does nothing but consume two clock cycles

BRK Break. Sets the B status bit to 1, and interrupts through the IRQ interrupt vector at $FFFE and $FFFF. The interrupt involves pushing the current address in the PC-register on the stack, followed by the current status register (with the B-bit set to 1, in this case); setting the I-bit to 1 to inhibit further IRQ interrupts; and loading the PC-register from $FFFE and $FFFF.

		Accumulator	Immediate	Zero Page	Absolute	Zero Page,X	Absolute,X	Zero Page,Y	Absolute,Y	(Zero Page,X)	(Zero Page),Y	
ADC	Add with Carry A <-- (A) + (M) + (C)	--	69 2	65 3	6D 4	75 4	7D *4	-- --	79 *4	61 6	71 *5	NV....ZC
AND	Logical "And" A <-- (A) and (M)	--	29 2	25 3	2D 4	35 4	3D *4	-- --	39 *4	21 6	31 *5	N.....Z.
ASL	Shift Byte Left C <-- [7.....0] <-- 0	0A 2	--	06 5	0E 6	16 6	1E 7	-- --	-- --	-- --	-- --	N.....ZC
BIT	Test Bits in Memory Z <-- (A) and (M), N <-- (M)bit 7, V <-- (M)bit 6	--	--	24 3	2C 4	--	--	--	--	--	--	NV....Z.
CMP	Compare with A-register N,Z,C <-- (A) - (M)	--	C9 2	C5 3	CD 4	D5 4	DD *4	-- --	D9 *4	C1 6	D1 *5	N.....ZC
CPX	Compare with X-register N,Z,C <-- (X) - (M)	--	E0 2	E4 3	EC 4	--	--	--	--	--	--	N.....ZC
CPY	Compare with Y-register N,Z,C <-- (Y) - (M)	--	C0 2	C4 3	CC 4	--	--	--	--	--	--	N.....ZC
DEC	Decrement Memory Byte M <-- (M) - 1	--	--	C6 5	CE 6	D6 6	DE 7	-- --	-- --	-- --	-- --	N.....Z.
EOR	Exclusive-Or A <-- (A) eor (M)	--	49 2	45 3	4D 4	55 4	5D *4	-- --	59 *4	41 6	51 *5	N.....Z.
INC	Increment Memory Byte M <-- (M) + 1	--	--	E6 5	EE 6	F6 6	FE 7	-- --	-- --	-- --	-- --	N.....Z.
LDA	Load A-register A <-- (M)	--	A9 2	A5 3	AD 4	B5 4	BD *4	-- --	B9 *4	A1 6	B1 *5	N.....Z.
LDX	Load X-register X <-- (M)	--	A2 2	A6 3	AE 4	--	--	B6 4	BE *4	--	--	N.....Z.
LDY	Load Y-register Y <-- (M)	--	A0 2	A4 3	AC 4	B4 4	BC *4	-- --	-- --	-- --	-- --	N.....Z.
LSR	Shift Byte Right 0 --> [7.....0] --> C	4A 2	--	46 5	4E 6	56 6	5E 7	-- --	-- --	-- --	-- --	N.....ZC
ORA	Inclusive-Or A <-- (A) or (M)	--	09 2	05 3	0D 4	15 4	1D *4	-- --	19 *4	01 6	11 *5	N.....Z.
ROL	Roll Byte Left C <-- [7.....0] <-- C	2A 2	--	26 5	2E 6	36 6	3E 7	-- --	-- --	-- --	-- --	N.....ZC
ROR	Roll Byte Right C --> [7.....0] --> C	6A 2	--	66 5	6E 6	76 6	7E 7	-- --	-- --	-- --	-- --	N.....ZC
SBC	Subtract with Borrow A <-- (A) - (M) + (C) - 1	--	E9 2	E5 3	ED 4	F5 4	FD *4	-- --	F9 *4	E1 6	F1 *5	NV....ZC
STA	Store A-register M <-- (A)	--	--	85 3	8D 4	95 4	9D 5	-- --	99 5	81 6	91 6	....
STX	Store X-register M <-- (X)	--	--	86 3	8E 4	--	--	96 4	--	--	--	....
STY	Store Y-register M <-- (Y)	--	--	84 3	8C 4	94 4	--	--	--	--	--	....

* Add one cycle if indexing causes a page boundary to be crossed.

Relative Branch Instructions

BPL	Branch if Plus	(Branch if N=0)	10	2 cycles if no branch;
BMI	Branch if Minus	(Branch if N=1)	30	
BVC	Branch if No Overflow	(Branch if V=0)	50	3 cycles if branch into same page;
BVS	Branch if Overflow	(Branch if V=1)	70	
BCC	Branch if Carry Clear	(Branch if C=0)	90	4 cycles if branch into different page.
BCS	Branch if Carry Set	(Branch if C=1)	B0	
BNE	Branch if Not Equal	(Branch if Z=0)	D0	
BEQ	Branch if Equal	(Branch if Z=1)	F0	

Jump Instructions

JMP	Unconditional Jump Absolute	4C	3
JMP	Unconditional Jump Indirect	6C	5
JSR	Jump to Subroutine	20	6

Implied Address Mode Instructions

BRK	Break (Set B=1,Generate IRQ Interrupt)		00	7	...B.I..
CLC	Clear Carry	C <-- 0	18	2	C
CLD	Clear Decimal Mode	D <-- 0	D8	2	....D...
CLI	Clear Interrupt Mask	I <-- 0	58	2	.....I..
CLV	Clear Overflow Status	V <-- 0	B8	2	.V......
DEX	Decrement X-register	X <-- (X) - 1	CA	2	N.....Z.
DEY	Decrement Y-register	Y <-- (Y) - 1	88	2	N.....Z.
INX	Increment X-register	X <-- (Y) + 1	E8	2	N.....Z.
INY	Increment Y-register	Y <-- (Y) + 1	C8	2	N.....Z.
NOP	No Operation	Does Nothing	EA	2	....
PHA	Push A-register on stack	M(S) <-- (A), S <-- (S) - 1	48	3	....
PHP	Push P-register on stack	M(S) <-- (P), S <-- (S) - 1	08	3	....
PLA	Pull Stack to A-register	S <-- (S) + 1, A <-- (M(S))	68	4	N.....Z.
PLP	Pull Stack to P-register	S <-- (S) + 1, P <-- (M(S))	28	4	restored
RTI	Return from Interrupt	Pull to P and PC	40	6	restored
RTS	Return from Subroutine	Pull to PC	60	6	....
SEC	Set Carry	C <-- 1	38	2	C
SED	Set Decimal Mode	D <-- 1	F8	2	....D...
SEI	Set Interrupt Mask	I <-- 1	78	2	.....I..
TAX	Transfer A to X	X <-- (A)	AA	2	N.....Z.
TAY	Transfer A to Y	Y <-- (A)	A8	2	N.....Z.
TSX	Transfer S to X	X <-- (S)	BA	2	N.....Z.
TXA	Transfer X to A	A <-- (X)	8A	2	N.....Z.
TXS	Transfer X to S	S <-- (X)	9A	2	....
TYA	Transfer Y to A	A <-- (Y)	98	2	N.....Z.

Chapter 9 -- SWEET-16

SWEET-16 is a powerful programming tool developed by Steve Wozniak in the early days of Apple. Chances are that you do have this tool, whether you know it or not. The standard version is hidden away inside the Integer BASIC system. If you have Integer BASIC on your mother board, or in a firmware card, or in a 16K RAM card, then you have SWEET-16. I have included a commented source file of SWEET-16 on your S-C MACRO ASSEMBLER II disk, so you can assemble your own copy if you wish.

SWEET-16 is really a language, just like 6502 machine language, BASIC, Pascal, FORTRAN. It looks a lot like a machine language for a computer that does not really exist, so "Woz" has called it his "dream machine". You can read all about it in an old issue of BYTE Magazine (November, 1977, pages 150-159): "SWEET-16 -- The 6502 Dream Machine". Another article you may want to find is "SWEET-16 Revisited", by Charles F. Taylor, in MICRO--The 6502/6809 Journal. January, 1982, pages 25-42.

The beauty of SWEET-16 is in its ability to perform 16-bit arithmetic and data moves using automatically updated address pointers. And to add icing to the cake, most of the instructions are only one byte long! You can write extremely compact code, if you are willing to pay the price of slower execution. (A typical program will take half as many bytes, but ten times longer to execute.)

Does anyone really use SWEET-16? Yes, in a big way. I used it in several places inside the early versions of S-C Assembler II. The TED/ASM assembler, and all its descendants (including DOS Tool Kit, TED II+, Big Mac, Merlin, and others) use SWEET-16 heavily. Several of the programs in the Apple Programmer's Aid ROM use SWEET-16. including the Integer BASIC Renumber/Append program.

The standard version of SWEET-16 is invoked by the 6502 instruction "JSR $F689"; the bytes immediately following contain opcodes for SWEET-16 to process. SWEET-16 opcodes will be executed until the "RTN" opcode, which returns to 6502 mode.

Programming Model

The SWEET-16 "machine" has sixteen 16-bit registers (R0-R15). R0 is actually the two memory bytes at $0000 and $0001. The next two bytes are called R1; R15 is stored in $001E and $001F. Several of the registers have special functions: R0 is used as an accumulator (like the 6502's A-register); R12 is the subroutine return stack pointer; R13 receives the results of comparisons; R14 is a status register; R15 is the program address counter.

SWEET-16 ...GISTERS

Register	6502 Address	Purpose
0	$00,01	Accumulator
1	$02,03	General
2	$04,05	General
.	.	.
.	.	.
.	.	.
11	$16,17	General
12	$18,19	Subroutine Stack Pointer
13	$1A,1B	Difference of comparands
14	$1C,1D	Status
15	$1E,1F	Program address

There are two general types of opcodes recognized by SWEET-16: register and non-register opcodes. The non-register opcodes all have the form "0x", where x is a hexadecimal digit from 0 through C. (Opcodes 0D, 0E, and 0F are not used.) These opcodes are used for relative branches, subroutine call and return, and to leave SWEET-16. The register opcodes have the format "xR", where x is a hexadecimal digit from 1 through F, and R is a register number (0-F).

SWEET-16 OPCODES

Non-Register Opcodes: RTN. BK, and RS are one byte opcodes. The rest have a second byte which is a relative address, similar to the relative branch addresses used in 6502 opcodes. The conditional branches use status bits found in R14.

00	RTN		Return to 6502 code.
01 ea	BR	addr	Unconditional Branch.
02 ea	BNC	addr	Branch if Carry=0.
03 ea	BC	addr	Branch if Carry=1.
04 ea	BP	addr	Branch if last result positive.
05 ea	BM	addr	Branch if last result negative.
06 ea	BZ	addr	Branch if last result zero.
07 ea	BNZ	addr	Branch if last result non-zero.
08 ea	BM1	addr	Branch if last result = -1.
09 ea	BNM1	addr	Branch if last result not -1.
0A	BK		Execute 6502 BRK instruction.
0B	RS		Return from SWEET-16 subroutine.
0C ea	BS	addr	Call SWEET-16 subroutine.

Register Opcodes: The SET opcode uses three bytes, to load a 16-bit immediate value into a register. All the rest of the register opcodes only use one byte. In this table, "MA" means Memory Address.

1n lo hi	SET	n,value	Rn <-- value.
2n	LD	n	R0 <-- (Rn).
3n	ST	n	Rn <-- (R0).
4n	LD	@n	MA = (Rn), R0L <-- (MA), Rn <-- MA+1, R0H <-- 0.
5n	ST	@n	MA = (Rn), MA <-- (R0L), Rn <-- MA+1.
6n	LDD	@n	MA = (Rn), Rn <-- (MA, MA+1), Rn <-- MA+2.
7n	STD	@n	MA = (Rn), MA,MA+1 <-- (R0), Rn <-- MA+2.
8n	POP	@n	MA = (Rn)-1, R0L <-- (MA), R0H <-- 0, Rn <-- MA.
9n	STP	@n	MA <-- (Rn)-1, MA <-- R0L, Rn <-- MA.
An	ADD	n	R0 <-- (R0) + (Rn).
Bn	SUB	n	R0 <-- (R0) - (Rn).
Cn	POPD	@n	MA = (Rn)-2, MA,MA+1 <-- R0, Rn <-- MA.
Dn	CPR	n	R13 <-- (R0) - (Rn), R14 <-- status flags.
En	INR	n	Rn <-- (Rn) + 1.
Fn	DCR	n	Rn <-- (Rn) - 1.

The S-C Assembler II includes all of the SWEET-16 opcodes, in the formats shown above. You can write programs which mix bot 6502 code and SWEET-16 together in any combination.

Here are a few examples which illustrate programming in SWEET-16.

```
                        1000 *--------------------------------
                        1010 *      CLEAR A BLOCK OF MEMORY
                        1020 *--------------------------------
F689-                   1030 SWEET.16   .EQ $F689
0A00-                   1040 BLOCK      .EQ $A00
0234-                   1050 N          .EQ $234
                        1060 *--------------------------------
0800- 20 89 F6 1070 CLEAR   JSR SWEET.16
0803- 10 00 00 1080         SET 0,0       0 FOR CLEARING WITH
0806- 11 00 0A 1090         SET 1,BLOCK   ADDRESS OF BLOCK
0809- 12 34 02 1100         SET 2,N       # BYTES TO CLEAR
080C- 51       1110 .1      ST  @1        STORE IN BLOCK
080D- F2       1120         DCR 2
080E- 07 FC    1130         BNZ .1        NOT FINISHED YET
0810- 00       1140         RTN

SYMBOL TABLE

0A00- BLOCK
0800- CLEAR
.01=080C
0234- N
F689- SWEET.16

0000 ERRORS IN ASSEMBLY
```

```
                    1000 *--------------------------------
                    1010 *       MOVE A BLOCK OF MEMORY
                    1020 *--------------------------------
F689-               1030 SWEET.16   .EQ $F689
0A00-               1040 SOURCE     .EQ $A00
0A80-               1050 DESTIN     .EQ $A80
0023-               1060 N          .EQ $23
                    1070 *--------------------------------
0800- 20 89 F6      1080 MOVE   JSR SWEET.16
0803- 11 00 0A      1090        SET 1,SOURCE  ADDRESS OF SOURCE BLOCK
0806- 12 80 0A      1100        SET 2,DESTIN  ADDRESS OF DESTINATION BLOCK
0809- 13 23 00      1110        SET 3,N       # BYTES TO MOVE
080C- 41            1120 .1     LD  @1        GET BYTE FROM SOURCE
080D- 52            1130        ST  @2        STORE IN DESTINATION
080E- F3            1140        DCR 3
080F- 07 FB         1150        BNZ .1        NOT FINISHED YET
0811- 00            1160        RTN
```

SYMBOL TABLE

```
0A80- DESTIN
0800- MOVE
.01=080C
0023- N
0A00- SOURCE
F689- SWEET.16
```

0000 ERRORS IN ASSEMBLY

```
                    1000 *--------------------------------
                    1010 *      RENUMBER S-C ASSEMBLER SOURCE CODE
                    1020 *--------------------------------
F689-               1030 SWEET.16   .EQ $F689
004C-               1040 HIMEM      .EQ $4C,4D
00CA-               1050 PP         .EQ $CA,CB
                    1060 *--------------------------------
                    1070 RENUMBER
0800- 20 89 F6      1080        JSR SWEET.16
0803- 11 CA 00      1090        SET 1,PP      PP HAS ADDRESS OF SOURCE CODE
0806- 61            1100        LDD @1        GET ADDRESS OF SOURCE CODE
0807- 31            1110        ST  1         ...IN R1
0808- 12 0A 00      1120        SET 2,10      INCREMENT = 10
080B- 13 4C 00      1130        SET 3,HIMEM   HIMEM HAS ADDRES OF END OF SOURCE
080E- 63            1140        LDD @3        GET ADDRESS IN HIMEM
080F- 33            1150        ST  3         ...IN R3
0810- 14 DE 03      1160        SET 4,990     START = 990 (FIRST LINE WILL BE 1000)
0813- 21            1170 .1     LD  1         TEST IF FINISHED
0814- D3            1180        CPR 3
0815- 03 0E         1190        BC  .2        YES
0817- 41            1200        LD  @1        GET # BYTES IN THIS SOURCE LINE
0818- 35            1210        ST  5         ...INTO R5
0819- 24            1220        LD  4         GET SEQUENCE NUMBER
081A- A2            1230        ADD 2         ADD INCREMENT
081B- 34            1240        ST  4         ...INTO R4 AGAIN
081C- 71            1250        STD @1        ...AND ALSO INTO SOURCE LINE
081D- F1            1260        DCR 1         BACK UP POINTER
081E- F1            1270        DCR 1
081F- F1            1280        DCR 1
0820- 21            1290        LD  1         ADD LENGTH OF LINE TO POINTER
0821- A5            1300        ADD 5
0822- 31            1310        ST  1         POINT AT NEXT SOURCE LINE
0823- 01 EE         1320        BR  .1
0825- 00            1330 .2     RTN
0826- 60            1340        RTS
```

SYMBOL TABLE

```
004C- HIMEM
00CA- PP
0800- RENUMBER
.01=0813, .02=0825
F689- SWEET.16
```

0000 ERRORS IN ASSEMBLY

OPERATION AND MEMORY USA

Configuration Requirements

S-C Macro Assembler will run in any Apple II or Apple II Plus with at least 32K RAM. You can assemble much larger source programs if you have 48K of RAM. You will need at least one standard Apple disk drive.

If you have a RAM card (16K RAM or larger) in slot 0, you can use the language card version of the assembler for still larger source programs.

Contents of the Disk

The disk you received with your S-C Macro Assembler is a standard 16-sector DOS 3.3 disk. It can be copied with Apple's disk copy programs, and the individual files are copyable with FID. (If you do not have DOS 3.3, the files can be de-MUFFINed; or, you can order a special copy on a 13-sector disk.)

There are two versions of the S-C Macro Assembler on the disk. S-C.ASM.MACRO is the standard version, which loads at $1000. S-C.ASM.MACRO.LC is the language card version; it loads at $D000 in any 16K RAM card.

The type "T" file named LOAD LCASM is a control (EXEC) file used to load the language card version.

The type "I" file named MACRO LIBRARY is an assembly language source file with some sample macro definitions. You may find them useful tools in your programming.

The rest of the type "I" files are sample assembly language source files. The assembler uses file type "I" for source program files. They will LOAD into Integer BASIC, but they will be meaningless there. True Integer BASIC files will LOAD into the S-C Macro Assembler, but they will be meaningless there. A brief description of the sample programs appears later in this Appendix.

Memory Usage

The standard version of the S-C Macro Assembler program is about 8704 bytes long ($2200), and occupies $1000 through $31FF in memory. The symbol table begins at $3200 and extends upward; your source program begins at the bottom of DOS ($9600 in a 48K machine) and extends downward.

The language card version of the S-C Macro Assembler program is a little longer than the standard version. It loads into the

language card at $D000. The EXEC file which loads the assembler into the language card also configures it so that DOS thinks of it as the alternate to the language in ROM on the mother board. The symbol table is set up to begin at $1000 rather than $3200.

During source program entry or editing, memory usage is monitored so that the source program does not grow so large as to overlap the symbol table. Overlapping will cause the "MEM FULL ERROR" message to print. During assembly, memory required by the symbol table is monitored, to prevent the symbol table from overlapping the source program. Overlapping will generate the "MEM FULL ERROR" message and abort the assembly.

In addition, memory usage by the object program is monitored, so that it will not destroy the source program, DOS, the S-C Macro Assembler, the symbol table, or switch any I/O addresses. Therefore, if the object program bytes are directed at any memory address between $1000 and the top of the symbol table, or any address above the beginning of the source program, the "MEM PROTECT ERROR" will be printed and assembly aborted.

If you are using macros with private labels, the private label table extends from $0FFF downward toward $0800. The private label table is also protected during assembly. Each private label uses five bytes in this table; the maximum number of private labels in an assembly is therefore 409.

The assembler also uses many locations in page zero during editing or assembly. Of particular importance are $4A-4D, $CA-CD, $73-74, and $D9. Location $D9 is used by DOS as a flag to allow commands to be entered. The other locations are used to point at the beginning and end of your source program and symbol table.

Locations $00 through $1F are not used at all by S-C Macro Assembler; you may use them as you wish without any fear of conflict.

Page one ($100-1FF) is used both as a stack and as storage for various items. The high addresses in page one are used for the stack. The low end is used for a symbol buffer and for pointers to the 27 hash chains used in storing the symbol table. The block from $170 through $1BF is used for holding search and replace strings by the editor, and for .TI titles during assembly.

Page two ($200-2FF) is used as an input buffer.

The high end of page three ($3D0-3FF) is used by DOS and by the assembler. You must not change any bytes between $3D0 and $3EF. $300-3CF is free to be used in any way you wish.

Locations $400-7FF are used by the Apple II as the screen buffer. There are 32 bytes which are unused by the screen image, which are instead used by certain peripheral boards such as the disk controller or printer interface boards.

Locations $800-$FFF are free to be used for storing your objec program, unless you are using private labels.

With care and planning, you can find space for object program storage between the top of the symbol table and the bottom of the source program. If the space there is too difficult to determine, or insufficient size, you need to use the ".TF" directive to store the object program directly on a binary dis file.

ROM USAGE

The S-C Macro Assembler takes full advantage of subroutines inside the Apple Monitor ROM. Here is a list of all the subroutines used:

F941	Print 4-digit hex value from A,X
F94A	Print (X) blanks
FB2F	Set text mode, full screen window parameters
FBF4	Advance cursor
FC10	Backspace cursor
FC1A	Move cursor up one line
FC22	VTAB to current CV value
FC2B	An RTS instruction
FC42	Clear to end of page
FC58	Clear screen, home cursor
FC66	Move cursor down one line
FC9C	Clear to end of line
FCA8	Delay
FD0C	Read next input character from keyboard
FD18	Read next input character through $38,39
FD84	Add char to input line
FD99	Print (Y,X) in hex with dash
FDDA	Print (A) in hex
FDED,FDF0	Print (A) as ASCII char
FE00	Display memory in hex
FE2C	Move block of memory
FE89	Set input to keyboard
FE93	Set output to screen
FECD	Write block of memory on tape
FEFD	Read tape into memory
FF2D	Print "ERR", ring bell
FF3A	Ring bell
FF69	Enter Monitor for MNTR Command
FFA7	Get hex number
FFBE	Process monitor command
FFC7	Clear monitor mode byte
FFCC	Table of monitor commands

*** VALUE > 255 ERROR	A local label is more than 255 bytes from its normal label.
*** NO NORMAL LABEL ERROR	A local label is used with no normal label present.
*** NESTED .IN ERROR	There is an .IN directive within an included file.
*** MISSING .DO ERROR	There is a .PIN or .ELSE without a corresponding .DO.
*** .DO NEST TOO DEEP ERROR	.DO - .PIN blocks are nested more than eight levels deep.
*** KEY TOO LONG	The search string in a command is longer than 38 characters.
*** REPLACE TOO LONG ERROR	The REPLACE command tried to create a line longer than 248 characters.
*** NO MACRO NAME ERROR	The .MA directive has no name in the operand field.
*** UNDEFINED MACRO ERROR	The macro name called has not been defined.
*** BAD MACRO PARAMETER ERROR	The character following a square bracket (]) must be a number (1-9) or a (0).

When an error is discovered during assembly, the error message is printed along with the offending line. The assembler then continues its pass, looking for more errors. At the end of the pass it will print "XXXX ERRORS IN ASSEMBLY", where XXXX is the number of errors it found in that pass. If there are any errors discovered during pass one, assembly will not continue into pass two. Some errors are catastrophic, and abort assembly without continuing to the end of the pass.

Appendix C

PRINTER SOFTWARE

If you have a standard Apple parallel or serial printer interface, with firmware in ROM on the card, you probably do not need any special printer software. You can turn your printer on from within the S-C Macro Assembler using the "PR#slot" command, and turn it off with "PR#0".

If you have a special interface, which requires non-standard setup not handled by the PR# command, you can use the PRT command. When you type the PRT command, the S-C Macro Assembler executes a "JSR $1009" instruction. At $1009 there is a JMP instruction, which you can patch to point at your special printer setup routine.

For example, if you have a special printer driver loaded at $300, your setup program might look like this:

```
                    1000 *--------------------------------
                    1010 *       SAMPLE PRINTER DRIVER
                    1020 *--------------------------------
                    1030         .OR $300
0300- A9 0C         1040 PRT     LDA #DRIVER
0302- 85 36         1050         STA $36
0304- A9 03         1060         LDA /DRIVER
0306- 85 37         1070         STA $37
0308- 20 EA 03      1080         JSR $3EA      DOS I/O REHOOK
                    1090 *  OTHER SETUP WOULD COME HERE
030B- 60            1100         RTS
                    1110 *--------------------------------
                    1120 DRIVER
                    1130 *
                    1140 *  WHATEVER IT TAKES TO TALK TO YOUR PRINTE
                    1150 *  GOES HERE
                    1160 *
030C- 60            1170         RTS
```

The S-C Macro Assembler Diskette includes a more extensive printer driver, written to use a serial interface connected to one of the output pins of the Apple II game connector.

Appendix D

CUSTOMIZING

A number of features within the S-C Macro Assembler have been parameterized, to make it easier for you to customize it the way you like.

The parameters are all grouped at the beginning of the program at $1006-101D in the normal version, and $D006-D01D in the language card version. You can adjust the parameter values as you like, and BSAVE the resulting personal version for a permanent copy.

$1000: Hard entry point. Performs major initialization. If you want some additional initialization you can modify the address of this JSR, and have your program end by JMP to the address which was in $1001,1002.

$1003: Soft entry point. If you desire some additional code here you can patch it in the same way as the hard entry above.

$1006: USR Vector. If you want to implement a USR command, change the address in this JMP instruction to call your command. Your command should end with an RTS, or a JMP $1003 ($D003 for language card version).

$1009: PRT Vector. If you want to implement your own PRT command, put the address of your command processor in bytes $100A and $100B. Your command should end as the USR command described above.

$100C: .US Directive Vector.

$100F: Tab Control Character: Normally $89, which is control-I. You can change it to some other control character if you wish.

$1010-1014: Tab Column Settings. Up to five tab stops. Normally set to 14, 18, 27, 32, and 0. If you wish fewer than five tab stops, use 0 for the unused ones.

$1015: Escape-L Comment Line Repeated Character. Normally $AD, which is "-". If you want a line of asterisks, use $AA. Use any printing character you want.

$1016: Lower-Case Mod Flag. Normally 0. Set to $FF if all three conditions are true:

1) you have installed a lower case display modification in your Apple;
2) you have installed a shift-key modification in your Apple (a wire from the shift-key solder terminal to pin 4 of the game connector);
3) you want to use lower-case in your source programs.

$1017: Compression flag. Normally $04, which means compression is ON. Set to $FF to turn compression OFF. When off, source files will be compatible with S-C Assembler II Version 4.0; when ON. they will not be. The compression switched by this flag replaces strings of repeated non-blank characters of 4 or more in length with a 3-byte token sequence. Multiple blank compression is not affected by this flag.

$1018: Search string wildcard character. Normally $17, which is control-W. (Note that the high-bit is clear.)

$1019: Character Output Vector. Normally contains "JMP $FDED", but you can use your own character output routine if you wish. Most output from the editor and assembler is vectored through this point.

$101C: Always $A9, first byte of LDA-immediate instruction.

$101D: Symbol Table starting page number. Normally $32 for normal version, $10 for language card version.

To patch the language card version it is necessary to write-enable the RAM card before making the changes.

```
]EXEC LOAD LCASM
:$AA60.AA61
AA60:XX YY
:$C083 C083
:$addr:value
 .... etc.
:RENAME S-C.ASM.MACRO.LC,LCASM.OLD
:BSAVE S-C.ASM.MACRO.LC,A$D000,L$YYXX
```

Appendix E

BIBLIOGRAPHY

There are now in print a number of good books and periodicals for learning how to program the 6502 microprocessor. Here are the ones I have found helpful.

Periodicals:

Apple Assembly Line, a monthly newsletter published by S-C SOFTWARE, See advertisement inside back cover for details.

Assembly Lines, a monthly column by Roger Wagner in SOFTALK magazine. 11021 Magnolia Blvd., North Hollywood, CA 91601

NIBBLE Magazine. Box 325, Lincoln, MA 01773

MICRO, the 6502/6809 Journal. P.O. Box 6502, Chelmsford, MA 01824

Call A.P.P.L.E. magazine. 304 Main Ave. S, Suite 300, Renton, WA 98055

Books:

6502 Software Design, Leo J. Scanlon. one of the Blacksburg Continuing Education Series, published by Howard W. Sams & Co. 1980. 270 pages, paper, $10.50.

6502 Assembly Language Programming, Lance A. Leventhal. Osborne/McGraw-Hill, Inc., 1979. $16.99, paper. Over 80 programming examples, tested on an Apple II. (I recommend thi one to the really serious programmer, and keep some on hand at $16.00 each.)

Programming & Interfacing the 6502, With Experiments, Marvin L De Jong. One of the Blacksburg Continuing Education Series, published by Howard W. Sams & Co., 1980. 414 pages, paper, $13.95.

6502 Software Gourmet Guide & Cookbook, Robert Findley. Scelb Publications, 1979. 204 pages, paper. $10.95. Includes listings of conversion routines, search and sort routines, and floating point routines.

6502 Games, Rodney Zaks, SYBEX. The third in the SYBEX series on programming the 6502. Includes listings of games in assembly language, of the type which are usually programmed in BASIC.

Wozpak II and Other Assorted Goodies. A collection of Apple I documentation published by the publishers of CALL A.P.P.L.E. It contains many useful programs in assembly language which ca be used and/or studied.

Practical Microcomputer Programming: the 6502, W. J. Weller, Northern Technology Books, 1980. 459 pages, $32.95, cloth. Includes a listing of a 6502 assembler and of a debugging package.

Programming a Micro-Computer: 6502, Caxton C. Foster, Addison-Wesley Publishing Company, 1978. 234 pages, $9.95. Oriented toward KIM-1, but has very good explanations and examples of machine language.

Real Time Programming -- Neglected Topics, Caxton C. Foster, Addison-Wesley Publishing Company, 1978. 190 pages, $8.95. Covers interrupt handling, I/O interfaces, synchronizing, sampling, closed-loop control, communication, and more. Concise, clear explanations by a good teacher.

Apple Machine Language, Don & Kurt Inman, Reston Books (Prentice-Hall), 1981. 296 pages, $12.95 paper, $16.95 cloth. For the ultimate beginner.

How to Program Microcomputers, William Barden, Jr., Howard W. Sams & Co., 1977. 256 pages, $8.95. Covers the 6502, 6800, and 8080 microprocessors.

Using 6502 Assembly Language, Randy Hyde, Datamost Inc., 1981. 283 pages, $19.95. An excellent book, although it promotes Randy's LISA Assembler.

Beyond Games: Systems Software for Your 6502 Personal Computer, Ken Skier, BYTE/McGraw-Hill, 1981. 432 pages, $14.95. Sounds a lot better than it is.

6502 Assembly Language Subroutines, Lance Leventhal. Osborne/McGraw-Hill, 1982. 550 pages, $12.95. Very good collection of useful techniques, includes several Apple-specific examples.

Assembly Language Programming for the Apple II, Robert Mottola. Osborne/McGraw-Hill, 1982. 143 pages, $12.95.

Assembly Lines: The Book, Roger Wagner. Softalk Publishing, 1982. 270 pages, $19.95. Excellent book for the beginner, collected from Roger's magazine columns. The best book so far.

INDEX